Tosetti, Matías Alejandro

Gestión y compresión de memoria flash de una WSN aplicada al seguimiento genético de ganado

Tesis para la obtención del título de grado de Ingeniero Eléctrico Electrónico

Director: Laprovitta, Agustín Miguel

GESTIÓN Y COMPRESIÓN DE MEMORIA FLASH DE UNA WSN APLICADA AL SEGUIMIENTO GENÉTICO DE GANADO

Facultad de Ingeniería

UNIVERSIDAD CATÓLICA DE CÓRDOBA

Autor:

Tosetti Matías Alejandro

Director:

Dr. Ing. Agustín M. Laprovitta

Co-director:

Ing. Juan L. Castagnola

2016

ACEPTACIÓN DEL TRABAJO FINAL

Universidad Católica de Córdoba

Facultad de Ingeniería

Carrera de Ingeniería Electrónica

Título: **Gestión y compresión de memoria flash de una WSN aplicada al seguimiento genético de ganado**

Autor: Tosetti Matías Alejandro

Calificación:……

Firma y Aclaración de Presidente de Mesa Examinadora

Firma y Aclaración de Vocal de Mesa Examinadora

Firma y Aclaración de Vocal de Mesa Examinadora

Córdoba, ……. de ……..…….. de 2016

A mi madre Patricia, a mi padre Luis y a mi hermana Natalia

TABLA DE CONTENIDO

AGRADECIMIENTOS

RESUMEN

Bajo la premisa de optimizar recursos, el presente trabajo final orientó sus esfuerzos en el desarrollo de un firmware de control para la gestión de la memoria flash de una red de sensores inalámbricos. La finalidad de esta última consiste en encontrar la línea genética de cualquier animal mamífero destinado a la producción ganadera, mediante la identificación del par madre-cría.

Se propusieron tres métodos para el almacenamiento de las muestras recolectadas durante el estudio y se implementó el que permitió una mayor autonomía de memoria en tres posibles escenarios. Finalmente, se plantearon dos algoritmos de compresión sin pérdida de información para determinar la factibilidad de incorporarlos y aumentar aún más la autonomía de la red.

ABSTRACT

Under the premise of optimizing resources, this paper focused its efforts on developing a control firmware for flash memory management of a wireless sensor network. The network purpose is to find the genetic line of any mammal animal intended for livestock production, by identifying the mother-breeding pair.

Three methods were proposed to store the samples collected during the study and was implemented the one which allowed the greater autonomy memory in three possible scenarios. Finally, two compression algorithms were raised to determine the feasibility of incorporating them and further increase the autonomy of the network.

TABLA DE FIGURAS

1.INTRODUCCIÓN

El Seguimiento Genético de Ganado consiste en la determinación del par madre-cría con el objetivo de estudiar la variación y la transmisión de rasgos por la herencia genética. La identificación de la línea genética de los animales permitirá un mayor control de la productividad ganadera [1].

El presente trabajo se desarrolla en el contexto de un proyecto iniciado en el Laboratorio de Comunicaciones y Redes de Sensores Inalámbricos de la Universidad Católica de Córdoba (LCRS-UCC). Este consiste en el desarrollo de una Red de Sensores Inalámbricos (WSN, por sus siglas en inglés) de topología en malla con comunicación unidireccional y constituida por nodos. Estos nodos son de dos tipos: nodos Madre, que irán sujetos del collar de cada hembra y que sólo transmiten; y nodos Hijo que irán sujetos del collar de cada cría y que sólo reciben. Todos los nodos tienen capacidad de procesamiento al disponer de un microcontrolador, pueden enviar y transmitir ondas de radiofrecuencia al contar con una antena y un transceptor, cuentan con una fuente de alimentación propia a batería, y una limitada capacidad de almacenamiento de información.

Dado que la energía disponible es escasa, la red fue concebida para permanecer la mayor parte del tiempo en estado inactivo y encenderse, recibir o trasmitir, procesar y almacenar muestras en intervalos regulares, en un lapso de tiempo reducido.

De forma sintética, el funcionamiento del sistema puede describirse de la siguiente manera: durante los intervalos activos, cada nodo Madre transmite, de manera secuencial para evitar colisiones, su número de identificación (*ID* único en la Red) a través de una trama por radio frecuencia. Estas tramas serán recibidas sólo por aquellos nodos Hijos que se ubiquen dentro de su radio de alcance. Así, irán recibiendo de a una por vez, censando su potencia y calculando la hora y fecha de cada recepción. Estos tres datos serán agrupados constituyendo lo que se denomina una "*muestra*". Cada *muestra* será filtrada de acuerdo a su potencia asociada, para descartar las de menor importancia. Una vez aprobadas, serán almacenadas de manera ordenada en la memoria flash de cada nodo Hijo. Finalmente, luego de almacenar todas las *muestras* generadas por cada nodo Madre ubicado en su cercanía, todos los nodos vuelven a su estado de suspensión y ahorro energético. Este proceso recibe el nombre de "*ciclo*" y se repite en intervalos de tiempo configurables. El conjunto de *muestras* recolectadas en las memorias de cada nodo Hijo, al final del estudio, constituirán la información primaria para analizar el comportamiento del ganado.

Lo anterior se ilustra en la figura N° 1, donde de color verde están los nodos Hijos que llegan a recibir la trama enviada por el 1° nodo Madre que transmite su número de *ID*. El resto de los nodos Madres están aguardando su turno de transmisión y los nodos Hijos rojos no están dentro del alcance pero están expectantes censando su entorno en búsqueda de tramas.

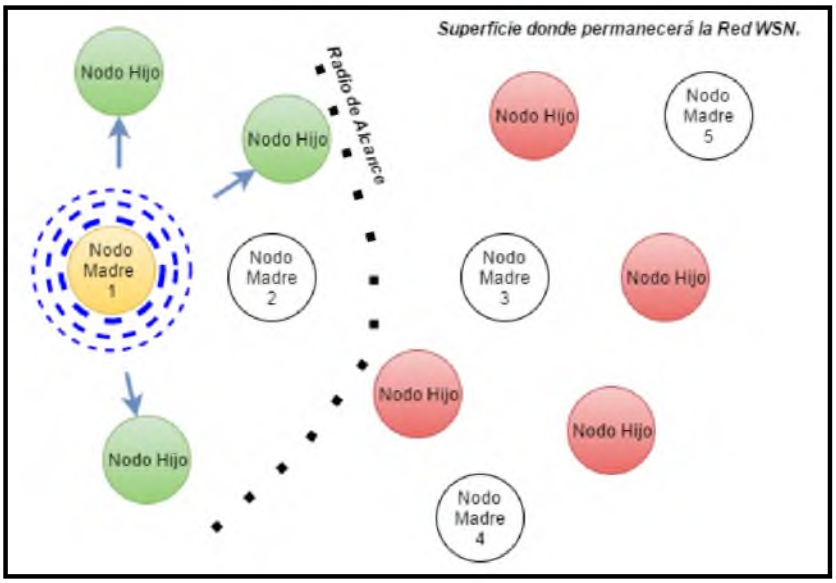

Figura 1: Ejemplo de funcionamiento de la red WSN bajo estudio.

La potencia censada de la trama recibida (cuyo nombre técnico es *RSSI*, Receiver Signal Strength Indication) es la que permitirá calcular la distancia relativa entre el nodo Madre y el nodo Hijo y encontrar estadísticamente la distancia de amamantamiento entre la hembra y la cría. Este evento permite determinar el par madre-cría ya que el amamantamiento de las crías es exclusivo con su madre [1].

Este trabajo final, a partir de la plataforma WSN desarrollada en el LCRS, desarrolló la sección del firmware de control que atiende la gestión de las *muestras* recibidas y su posterior almacenamiento en la memoria flash de cada nodo Hijo. Optimizar el espacio disponible en la memoria, elemento crítico cuya autonomía dictará la duración del estudio y el colapso de la red, fue el objetivo primario. Una mayor recolección de *muestras* permitirán que el estudio estadístico, para la determinación del par madre-cría, obtenga resultados más acertados.

La gestión de la memoria se llevó a cabo en cuatro pasos. El primero consistió en el estudio y programación de los procesos de escritura, lectura y borrado para crear una librería de manejo de hardware que dote al nodo con la capacidad básica de manipular de a bytes su memoria. El segundo paso fue delimitar un espacio en memoria para el guardado de *muestras* y cuyo uso no comprometa otras secciones empleadas para la ejecución de otros procesos. El tercer paso consistió en encontrar y programar, empleando los servicios de la librería de

hardware, un método que optimice el almacenamiento de *muestras*. El conjunto de algoritmos que ejecutan el método elegido conforman la librería de gestión de memoria. En el cuarto paso, se implementó el filtrado de *muestras* de acuerdo a su relevancia y se estudió la viabilidad de implementar algoritmos de compresión sin pérdida para concluir si su incorporación permite una mayor autonomía en el almacenamiento.

Este trabajo se organizó de la siguiente manera. En el capítulo 2 se inicia con una explicación del funcionamiento básico de una red WSN; en el capítulo 3 se detallan los principales elementos del nodo inalámbrico sobre el cual se trabajó; en el capítulo 4 se aborda el funcionamiento de la memoria flash integrada en cada nodo; en el capítulo 5 se analizan diferentes métodos de gestión de memoria y se explican los servicios ofrecidos por la librería creada a partir del método elegido; en el capítulo 6 se analizan diferentes algoritmos de compresión sin pérdida; en los capítulos 7 y 8 se aborda el impacto ambiental y la responsabilidad social universitaria respectivamente; y en el capítulo 9 se finaliza con una conclusión del trabajo abordado. Todas las referencias se enlistan en el capítulo 10.

1.1.OBJETIVOS

1.1.1.OBJETIVO GENERAL:

Realizar contribuciones al diseño de la red WSN desarrollada por el laboratorio LCRS-UCC para el Seguimiento de la línea genética de ganado. Estas se orientarán al aumento de autonomía de la memoria flash disponible en los nodos Hijos, a partir del desarrollo de un firmware de gestión de memoria flash.

1.1.2.OBJETIVOS ESPECÍFICOS:

- Crear algoritmos de manejo de flash que sólo puedan borrar, guardar y leer un espacio específico asignado de memoria. Estos constituirán la librería de manejo de hardware.
- Analizar, calcular y evaluar la conveniencia de diferentes métodos de gestión de memoria.
- Implementar algoritmos de gestión de memoria a partir del método seleccionado y orientados a la necesidad de la aplicación principal, el seguimiento genético de ganado. Estos conformarán la librería de gestión de *muestras*.
- Ofrecer la alternativa de filtrar *muestras* como un recurso para aumentar la autonomía de memoria.
- Analizar la factibilidad del uso de algoritmos de compresión.

2.INTRODUCCIÓN AL SISTEMA WSN

Las WSN se basan en dispositivos de bajo costo y mínimo consumo de energía, llamado genéricamente nodos. Estos son capaces de obtener información de su entorno (a través de sensores), procesarla localmente, y comunicarla a través de enlaces inalámbricos hasta un nodo central de coordinación, llamado concentrador o estación base. Los nodos además actúan como elementos de la infraestructura de la red al reenviar los mensajes transmitidos por nodos más lejanos hacia el concentrador [2]. La figura 2 muestra un esquema con los distintos elementos que forman parte de una plataforma WSN.

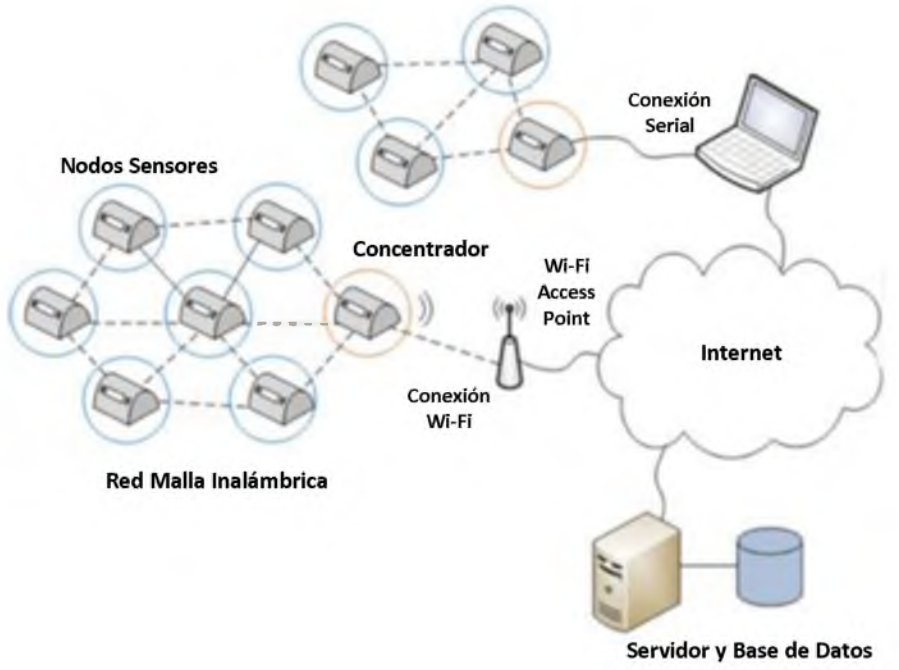

Figura 2: Esquema conceptual de una Red de Sensores Inalámbricos.

Dependiendo de las necesidades de la aplicación, la red está típicamente compuesta por un gran número de nodos sensores multifunción que son desplegados en una región o área de interés, o muy cerca de la misma [3]. Los mismos pueden comunicarse en distancias cortas, de allí que la colaboración mutua es esencial para realizar una tarea en común, como el monitoreo ambiental o el control de procesos industriales [4], [5].

La red está típicamente organizada en una de las tres topologías básicas, mostradas en la figura 3. En la topología de estrella, cada nodo sensor se conecta directamente al concentrador. Este es el arreglo más simple de implementar, sin embargo, no es eficiente para abarcar grandes superficies. En el caso de la topología de árbol, cada nodo se conecta a un nodo de mayor jerarquía en la estructura de la red y después al concentrador. Los datos son ruteados desde el nodo de menor jerarquía hasta el concentrador a través de un camino fijo.

Esta topología atiende correctamente los problemas de cobertura de la anterior, pero es muy poco tolerante a la falla de alguno de sus componentes.

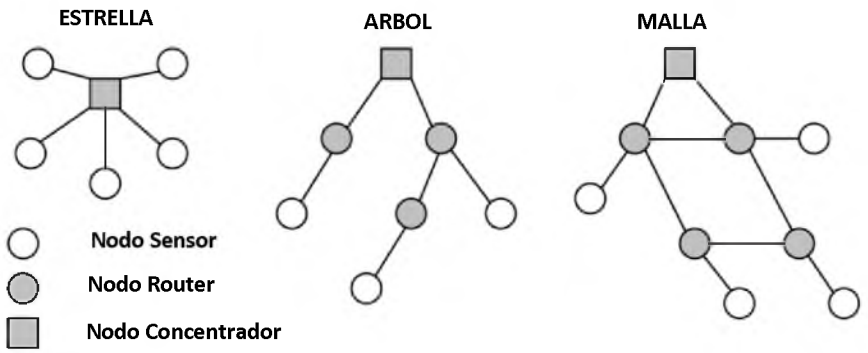

Figura 3: Topologías más comunes en WSN.

Finalmente, para ofrecer mayor confiabilidad y prestaciones, están las redes tipo malla. La característica de esta topología es que los nodos se pueden conectar a múltiples nodos en el sistema y pasar los datos por el camino disponible de mayor conveniencia conforme a los criterios específicos de la aplicación. En este caso es necesario que ciertos nodos de la estructura cumplan funciones de enrutadores. Esta topología requiere implementar complejos protocolos y mecanismos para la tarea de sincronización, acceso al medio y enrutamiento, lo que requiere nodos de mayores prestaciones y menor eficiencia energética.

Los nodos en la red pueden ser fijos y móviles y son unidades autónomas formadas típicamente por un microcontrolador (u otro dispositivo con capacidad de procesamiento), un transceptor de RF, uno o varios elementos sensores (con sus Interfaces analógicas y digitales) y una fuente de energía, que casi siempre es una batería (ver figura 4). Debido a las limitaciones de la batería, los nodos se diseñan priorizando la máxima conservación de la energía. Generalmente pasan la mayor parte del tiempo en modo inactivo con muy bajo consumo de potencia.

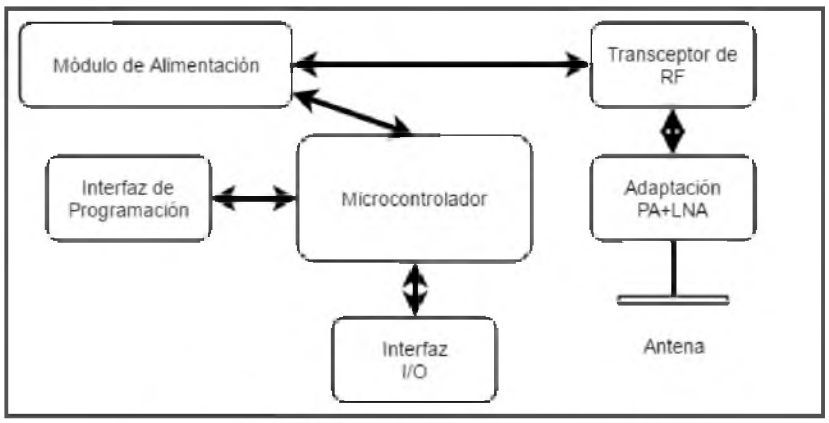

Figura 4: Arquitectura típica de un nodo WSN.

A medida que pasa el tiempo, el nodo sensor gradualmente consume la energía limitada. Finalmente, la energía se agota, disminuyendo la calidad del monitoreo de la red o incluso causando la paralización de la misma. Por lo tanto, el uso eficiente de la energía de todos los nodos para extender la vida útil de la red, es la principal preocupación en las WSN. De hecho, la mayoría de los estudios científicos en esta área está dedicada a esta problemática, y de allí surgen diferentes métodos y protocolos que influyen directamente sobre el consumo de la energía de los nodos [6], [7]. Otra solución, aunque no aplicable en todos los casos, es dotar al nodo de algún elemento de obtención de energía [8].

3.PLATAFORMA

A continuación se presentarán los elementos que componen a los nodos de la WSN bajo estudio. Estos nodos, mediante su configuración por firmware, pueden actuar como nodos Hijos, como nodos Madres y como nodos concentradores.

El microcontrolador MSP430F2254 de Texas Instruments, de ultra bajo consumo, con 16K bytes + 256 bytes de memoria flash y 512 bytes de memoria RAM embebidas, le brinda capacidad de procesamiento y de almacenamiento a cada nodo de la red. Mediante su puerto SPI maneja un transceptor, modelo CC2500 y del mismo fabricante, que a 2.4 Ghz y con muy bajo consumo permite que el nodo pueda enviar y recibir señales de radio frecuencia a través de una antena dipolo plegada. Además, censa la potencia (*RRSI*) de las señales recibidas, permitiendo obtener indirectamente uno de los datos más importantes: la distancia entre nodo Madre transmisor y el nodo Hijo receptor.

Una batería de Lithium –Polymer de 850mA alimenta a todos los componentes y puede ser recargada al finalizar cada estudio, mediante el módulo de carga Mircochip MCP73831. Un indicador lumínico led rojo informa al usuario del estado de la batería.

Un puerto serial JTAG permite programar al microcontrolador y depurar el código durante su ejecución. Finalmente, dos indicadores led y un pulsador brindan una interfaz al usuario. En la figura N° 5 puede observarse los componentes antes mencionados como así también su interrelación.

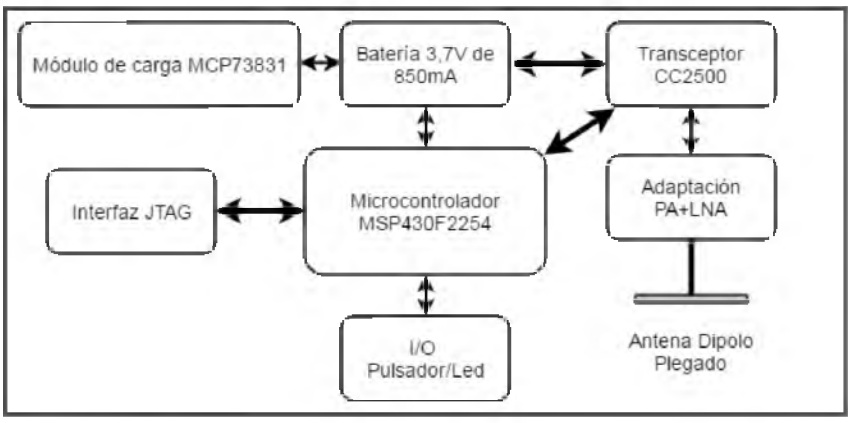

Figura 5: Diagrama de bloques de un Nodo WSN.

4.MEMORIA FLASH

4.1.INTRODUCCIÓN

En la presente sección se profundizará sobre la memoria flash del microcontrolador MSP430F2254, al ser el recurso que se utiliza para el almacenamiento de *muestras* en la red WSN bajo estudio.

¿Qué son las memorias flash? Son memorias pasivas de semiconductores de tecnología MOS que no poseen partes en movimiento. Las mismas se caracterizan por ser de acceso aleatorio, reprogramables, totalmente borrables eléctricamente y no volátiles, haciéndolas idóneas para el almacenamiento permanente de las instrucciones a ejecutarse por el microcontrolador y las *muestras* recolectadas durante el estudio de seguimiento genético de ganado.

El acceso a memoria es la acción de introducir información (escribir) u obtener (leer) aquella que se encuentre almacenada en una determinada posición (la información se almacena en grupo de bits accesibles simultáneamente y que se denominan posiciones). Este acceso puede ser directo o secuencial. Aquellas a las que no es necesario especificarles una posición (acción que se denomina direccionar) para efectuar un acceso, se denominan memorias de acceso secuencial. Los datos se introducen o escriben secuencialmente por el terminal de entrada, que tiene asignado la posición inicial de la memoria, y se extraen o leen por los terminales de salida cuya posición es la mayor dentro de la memoria. Por el contrario, las que pueden direccionarse para efectuar un acceso a cualquier posición, se denominan memorias de acceso directo. Dentro de esta última categoría, se encuentran las memorias flash.

La estructura interna es la manera en que se conectan entre sí las diferentes posiciones de memoria. La memoria flash cuenta con una estructura de acceso aleatorio o RAM (Random Access Memory por sus siglas en inglés). Éstas se caracterizan por ofrecer un tiempo de acceso único para cualquier posición.

Las memorias RAM pueden ser activas o pasivas de acuerdo a la permanencia de la información, parámetro que mide cualitativamente la diferencia entre el tiempo de lectura y el tiempo de escritura. En las activas, los tiempos necesarios para un acceso a escritura o a lectura son del mismo orden de magnitud. La memoria de trabajo temporal del microcontrolador MSP430F2254 es una memoria RAM activa de 512 bytes cuya información se borra automáticamente cuando se retira la tensión de alimentación. Habiendo diferentes

tipos de memorias de acceso aleatorio, ésta se denomina inapropiadamente memoria RAM por todos los fabricantes de circuitos integrados. Este texto no hará una excepción por razones de practicidad.

Los tiempos de acceso a escritura y a lectura de las memorias RAM pasivas difieren enormemente, siendo el primero, el mayor. Además, se distinguen por ser no volátiles, es decir, que no pierden la información almacenada aunque deje de aplicarse la tensión de alimentación.

Las memorias pasivas pueden ser totalmente pasivas o ROM (memorias de sólo lectura por sus siglas en inglés) que vienen escritas de fábrica y sólo pueden leerse; pasivas programables o PROM (memorias programables de sólo lectura por sus siglas en inglés) que pueden ser escritas una sola vez; y las pasivas reprogramables o RPROM (memorias reprogramables de sólo lectura por sus siglas en inglés) que pueden escribirse varias veces. En ésta última categoría encontramos las memorias EPROM (memorias programables borrables de solo lectura) que se borran por rayos ultravioletas; las EEPROM (memoria programable borrable eléctricamente de sólo lectura) que borra sólo de a una posición por vez y las memorias flash bajo estudio, borrables totalmente eléctricamente y, en contraste a las EEPROM, con velocidades de funcionamiento superiores debido a la capacidad de acceder a múltiples posiciones de memoria con una misma operación [9].

Un controlador gestiona los procesos de escritura y borrado de la memoria flash. Los módulos principales del controlador de memoria flash de las familias de microcontroladores MSP430F22XX son:

- 4 registros de control (FCTL1, FCTL2, FCTL3, FCTL4)
- Generador de tiempos (Timing Generator)
- Generador de voltaje programable (Proframming Voltage Generator)
- Latch de dirección (Address Latch)
- Latch de datos (Data Latch)
- Arreglo de memoria flash de silicio (Flash Memory Array)

En la imagen N° 6 podemos observar su diagrama de bloques general donde se aprecia la interconexión de los diferentes módulos. Las siglas MAB hacen referencia al bus de address a través del cual llegará la posición de memoria que se quiere acceder y las siglas MDB al bus por el cual se introducirán o se recogerán los datos. Ambos bus, además, permitirán direccionar y acceder a los cuatro registros de control del controlador. Los registros latch de dirección y el latch de datos le permiten al controlador retener temporalmente la posición y la información que será almacenada o que ha sido leída, para que la operatoria de

escritura, borrado y lectura se ejecuten de acuerdo al generador de tiempos y que el voltaje se establezca o se discontinúe por el generador de voltaje. El MAB, el MDB, el latch de dirección y el latch de datos son trasparentes al programador y sólo se mencionan para brindar un panorama general del funcionamiento interno del controlador de memoria flash.

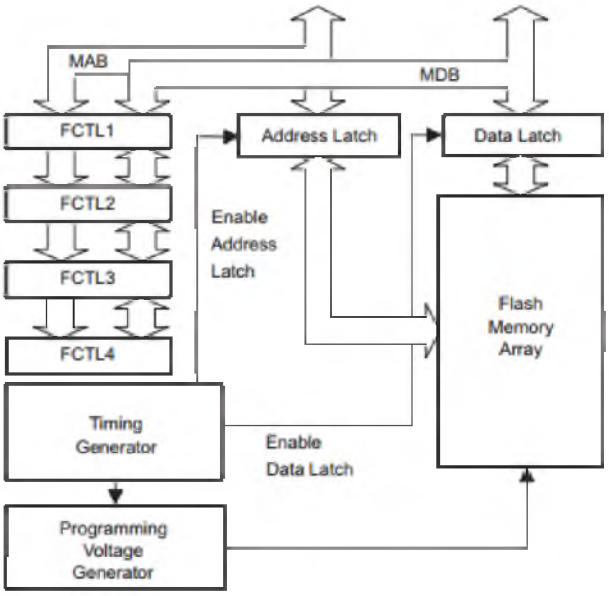

Figura 6: Diagrama de Bloques del Controlador de Memoria Flash de la familia MSP430.

4.2.SEGMENTACIÓN

El módulo de **arreglo de memoria flash** es la memoria de silicio propiamente dicha que se encuentra seccionada en dos:

- Memoria Principal (Main Memory)
- Memoria de Información (Information Memory)

El modo de funcionamiento de ambas es el mismo y se puede almacenar tanto datos como código. La diferencia radica en el direccionamiento físico y en su tamaño que varía de acuerdo al microcontrolador.

La Memoria Principal y la Memoria de Información están segmentadas en paquetes de 512 bytes y 64 bytes respectivamente. La cantidad de segmentos dependerá del tamaño de la secciones. Cada segmento de la Memoria Principal está subdividido en 8 bloques de 64 bytes. A modo de ejemplo, en la imagen 7 podemos observar la segmentación de un arreglo de memoria flash de 32k bytes de Memoria Principal y 512 bytes de Memoria de Información. En el microcontrolador MSP430F2254, la sección de Memoria Principal inicia en la dirección

0x0C000 y finaliza en 0x0FFC0. Estos son los límites dentro de los cuales se pueden guardar instrucciones, cómo el firmware del nodo, y datos, como las *muestras* recolectadas durante el estudio.

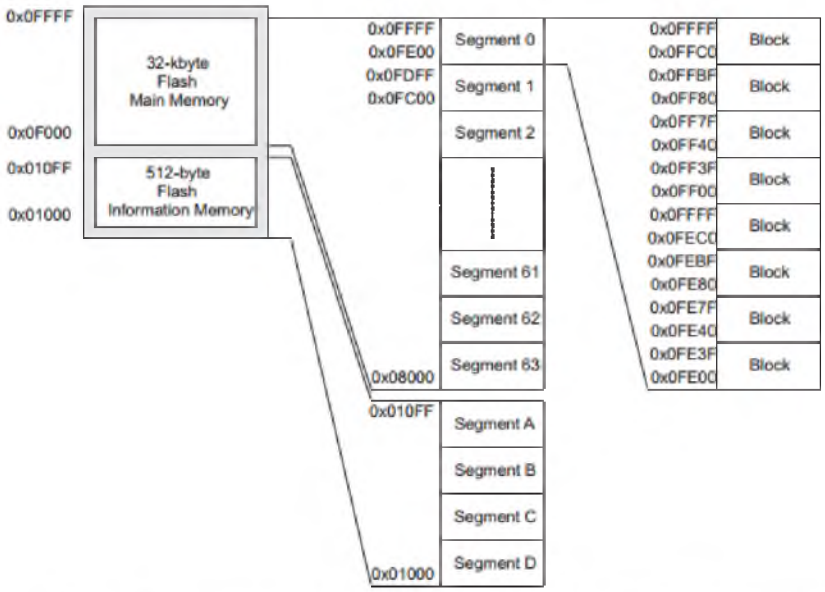

Figura 7: Segmentación de Memoria Flash de 32kb del microcontrolador MSP430F2274.

El segmento A de la Memoria de Información, almacena *parámetros* que usa el microcontrolador para inicializarse. Por recomendación del fabricante, este no se usa. El segmento B se destinó al registro de información interna a la red WSN. Para la familia MSP430F225X los segmentos B, C y D se encuentran entre las direcciones 0x010FF y 0x01000.

Esta segmentación, como sus direcciones, son importantes para comprender las operaciones de escritura y borrado que se explicarán en la siguiente sección. Estas operaciones se controlan por el **módulo de generación de tiempos** de la figura N° 8.

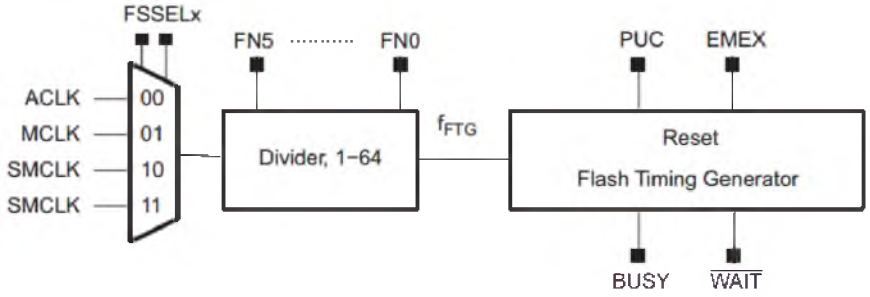

Figura 8: Módulo Generador de Tiempos de un controlador de memoria flash.

Éste se programó a través de los 4 registros (FCTL1, FCTL2, FCTL3, FCTL4) del controlador de memoria flash. La selección de la frecuencia de operación es importante debido a que debe encontrarse dentro del rango seguro especificado en la hoja de datos del

fabricante (Para nuestro caso es 257Khz – 476Khz). Para cualquier desvió de ese rango, no se asegura un correcto funcionamiento. Se programó para que la frecuencia principal sea la frecuencia del ciclo de instrucción provista por el Master Clock (MCLK) de 16Mhz. Se usó un prescaler de 46 para reducir la frecuencia a 333Khz y así trabajar con un valor intermedio del intervalo recomendado.

El **módulo de generación de voltaje** genera el voltaje de operación propicio para cualquier tipo de operación, sin necesitar proveer el mismo de manera externa.

4.3.OPERACIONES

El modo de operación por defecto de la memoria es el de lectura. Los módulos de generación de tiempos y voltajes se encuentran apagados y su funcionamiento se asemeja a una ROM (Read Only Memory Memoria de sólo lectura).

Mediante los 4 registros de control (FCTL1, FCTL2, FCTL3, FCTL4) se pueden elegir los siguientes modos de operación:

- Escritura de a byte o de a palabra (2 bytes)
- Escritura de a Bloque (64 bytes)
- Borrado de Segmento (512 o 64 bytes de acuerdo a la sección)
- Borrado Masivo (Todos los segmentos de la Memoria Principal)
- Borrado Total (Todos los segmentos, tantos los de la Memoria Principal como los de la Memoria de información excluyendo el Segmento A por defecto)

Si éstas son ejecutadas desde la memoria flash, durante el proceso, el CPU del microcontrolador queda en un bucle infinito sin procesar ninguna instrucción o interrupción. Estas últimas se deshabilitan automáticamente si los bits EEI y EEIEX del registro de la flash están en cero, para ejecutarse ni bien termine el proceso. En caso de no estarlo, ante una interrupción, se abortará el acceso a memoria y se producirá un mensaje de error a través de la flag de violación de acceso ACCVIFG y se procederá a atender a la misma. En caso de que se quiera ejecutar instrucciones durante la operación, ésta deberá realizarse desde la RAM. En este trabajo se realizó desde la misma memoria flash por lo que se obviará la explicación de los procesos necesarios para hacerlo desde la RAM. Además, se configuró para deshabilitar las interrupciones durante cualquier operación en flash. Guardar la información es de vital importancia en esta aplicación.

3.4.1.OPERACIÓN DE BORRADO

Por defecto, el módulo de arreglo de memoria flash está borrado con un nivel lógico igual a 1 en cada una de sus celdas básicas (1 bit). Para almacenar un nivel lógico 0, simplemente se lleva la celda de memoria a 0, pero para cambiar de un nivel lógico 0 a un 1, debe llevarse a cabo una operación de borrado. De esa manera, la ejecución de un borrado es necesario ya sea para eliminar información almacenada como para regrabar una determinada sección.

La unidad mínima que se puede borrar es 1 segmento de 512 bytes o 64 bytes de acuerdo a la sección de memoria. También se puede llevar a cabo un borrado masivo de toda la Memoria Principal o un borrado total donde ambas secciones vuelven a su estado por defecto (1 lógico).

La operación se resume en realizar lo que se denomina una "escritura basura" en cualquier parte del segmento a borrar, luego de configurar los registros apropiados. Los tiempos y el voltaje pasan a ser regulados por el generador de voltaje y por el generador de tiempos del controlador flash, y el CPU frena sus ejecuciones. Una vez finalizado el proceso, inicia desde la última instrucción o atiende, en orden de prioridad, las interrupciones que hayan sucedido.

La librería de manejo de hardware (o Hal por Hardware Abstraction Layer) desarrollada para este trabajo, lleva a cabo una operación de borrado mediante la siguiente función: **void erase (unsigned char *flash)**, cuyo argumento permite direccionar el segmento de memoria que se va a eliminar.

3.4.2.OPERACIÓN DE ESCRITURA

La escritura en memoria puede ser de a byte (8 bits), de a palabra (16bits) o de a bloque. Dado a que la aplicación de este trabajo no requiere un almacenamiento masivo en cada *ciclo*, el de a bloque no es necesario. Para flexibilizar el grabado de cantidades pares e impares, sólo se grabará de byte, prescindiendo del grabado a palabra. Éste es el que más consume y más tiempo de procesamiento conlleva debido a que los módulos de generación de voltaje y de tiempos deben ser inicializados por el controlador por cada byte a escribirse.

La librería Hal lleva a cabo esta operación mediante la función **void write_data_inFlash (uint16_t data_length, uint8_t sample_buff[], uint16_t flash_byte_position).** En ella deben ingresarse la longitud que va a grabarse, la posición donde se va a almacenar y la información que quiera guardarse. Por más que se especifique una longitud mayor a 1 byte, la escritura será de a byte y el proceso se repetirá tantas veces como sea especificado.

4.4.REGISTROS

Los registros del controlador flash (FCTL1, FCTL2, FCTL3, FCTL4) son de 16 bits y deben ser accedidos mediante el ingreso de la clave 0xA5 en los 8 bits más significativos del mismo. Un error en esta clave pone en 1 a la flag KEYV y dispara un reseteo PUC (por sus siglas en ingles de Power-up clear). Estos registros ordenan al controlador de memoria la ejecución de un determinado proceso. De esta manera, cada servicio de la librería Hal configura los 4 registros para ejecutar la operación que le corresponde. Finalmente, a través de estos registros, se programa el módulo de generación de tiempos.

5.GESTIÓN DE MUESTRAS EN MEMORIA FLASH

Dado a la baja disponibilidad de memoria para recolectar la suficiente cantidad de *muestras* válidas, que permitan identificar el par madre-cría, se analizaron algoritmos de gestión de memoria pensando en atenuar esta limitante.

Previo a la creación del algoritmo, fue necesario adoptar consideraciones iniciales que se enlistan a continuación:

- El tiempo es una variable muy importante y registrarlo en necesario para el análisis comportamental.
- El número de identificación (*ID*) de cada nodo debe almacenarse sin pérdidas.
- Las *muestras* deberán respetar un formato único de longitud no variable y medible en bytes para reducir el procesamiento.

5.1.CONEPTOS CLAVES PARA LA GESTIÓN DE LA MEMORIA

Varios conceptos serán introducidos en esta sección ya que son palabras claves en la gestión de la memoria que se abordará en las secciones 5.2 y subsiguientes.

El ***timestamp*** es un registro de 32 bits (4 bytes) que está dividido en dos partes de 16 bits. La parte baja es el registro del timer A (TAR, Timer A register) y la parte alta es el contador de rebasamientos del TAR. Éste número es un valor de referencia temporal común a todos los nodos, una vez sincronizados. A partir del mismo se puede inferir la fecha y la hora en la que la red WSN se sincronizó y empezó con sus rutinas programadas. Cada *muestra* debe tener uno asociado para que al final del estudio estén referenciadas temporalmente para el análisis estadístico.

Las ***muestras*** son un conjunto de 3 bytes que ensamblan dos elementos recibidos por el transceptor de RF CC2500:

- ***Número de Identificación ID*:** es un número que identifica de manera unívoca a cada nodo de la red WSN. Se destinaron 2 bytes para poder alcanzar al pequeño, mediano y gran productor al permitir 65536 nodos o cabezas de ganado.

- ***RSSI (por sus siglas en inglés: Received Signal Strength Indicator):*** Indicador de potencia de la señal recibida por el transceptor CC2500 en decibelios (db) con ½ db

de resolución. Consiste en un byte que se obtiene de su registro de estado luego de cada recepción. Su valor va de 0 a -64db, representando una pérdida de potencia, y de 0 a +63db, representando una ganancia. Este valor es usado para calcular la distancia relativa entre el nodo receptor y el trasmisor, es decir, entre cada animal.

Dado a que la unidad mínima en una operación de escritura es el byte, se trabajó con tamaños de *muestra* y *timestamp* múltiplos de uno. Además, el procesamiento conllevaría una complejidad inconveniente si se trabajara en bits. Como no se almacenará más de una decena de bytes por vez, la escritura de a bloque de 64 resultaría ineficiente en términos energéticos y perjudicial para la memoria temporal de datos RAM. Por último, la grabación de a palabra no fue tenida en cuenta debido a que la mayoría de las veces se registrará un número impar de bytes.

Un ***ciclo*** es aquel proceso que inicia con todos los nodos levantándose de su estado inactivo, para continuar con una secuencia de tareas que se ejecutan en el siguiente orden: los nodos Madres trasmiten de manera secuencial su número de identificación a través de tramas y por radio frecuencia; los nodos Hijos están constantemente censando su ambiente en búsqueda de estas tramas; en cada recepción, los nodos Hijos arman la *muestra* con el Id recibido, el *timestamp* calculado y el *RSSI* de la trama recibida; además, filtran las *muestras* de acuerdo a su *RSSI*; una vez finalizado el tiempo programado de recepción de tramas, el nodo Hijo procede a almacenar las *muestras* en memoria flash; finalmente, la red WSN vuelve a su estado de suspensión y ahorro energético. Un ***ciclo vacío*** o un ***ciclo sin muestras*** es aquel en cual el nodo Hijo no recibe ninguna trama y, por ende, no tiene *muestras* para almacenar.

El ***byte libre*** es la denominación usada para el primer byte vacío en la memoria asignada a *muestras* y donde deberá realizarse la siguiente grabación. Inicialmente, este estará en la posición cero y se irá desplazando a medida que se va llenando la memoria. Los algoritmos de gestión de memoria implementados, basan su funcionamiento en encontrarlo. Un byte vacío es equivalente a uno con todos sus bits en 1 lógico, valor por defecto de cada celda de memoria flash. Así, ningún byte a almacenarse podrá ser 0xFF ya que podría confundirse con uno vacío.

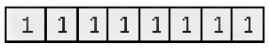

Figura 9: Byte vacío.

Los ***parámetros*** consisten en datos que necesita la red WSN para su funcionamiento y que no pueden perderse. Estos son almacenados dentro del segmento B de la Memoria de Información. Los mismos son el tipo de nodo (Madre o Hijo), el *offset* de calibración para la sincronización de toda la red, y los valores *RSSI* máximo y mínimo que representa la situación de amamantamiento.

5.2.ASIGNACIÓN DE MEMORIA FLASH PARA MUESTRAS Y PARÁMETROS

La asignación de la memoria es de vital importancia para no sobre escribir aquella que es empleada para código. De esta manera se usó la instrucción de pre-compilación **#pragma LOCATION** que ubica una variable de cualquier longitud en la dirección de memoria especificada. Este espacio quedará asignado a la variable y no podrá ser empleado por el compilador para guardar ninguna instrucción. Esta directiva es válida para el compilador CGT (Code Generation Tools) versión v4.x. o superior, usado por cualquier IDE CCS v5.x o superior.

Asignación de memoria al guardado de *parámetros* de la WSN:

```
#define FLASH_PARAMETER_ORIGIN 0x01080
#define PARAMETER_SEGMENT_SIZE 64

#pragma LOCATION (memory_parameter_segment, FLASH_PARAMETER_ORIGIN);
Uint8_t  memory_parameter_segment [PARAMETER_SEGMENT_SIZE];
```

Así, se reserva un buffer en memoria denominado **memory_parameter_segment[]**, desde la posición 0x01080 hasta la 0x010C0, es decir, uno de 64 bytes.

Asignación de memoria al guardado de *muestras*:

```
#define FLASH_SAMPLE_ORIGIN 0xC000
#define MEMORY_SAMPLE_SIZE (512*7)

#pragma LOCATION (memory_segment, FLASH_SAMPLE_ORIGIN);
Uint8_t memory_segment [MEMORY_SAMPLE_SIZE];
```

Así, se reserva un buffer en memoria denominado **memory_segment[]**, desde la posición 0x0C000 y de un tamaño de 3584 bytes. El resto de la memoria fue destinada a guardar el código a ejecutarse por el procesador para llevar a cabo las demás rutinas del nodo en la red WSN.

5.3.FILTRADO DE MUESTRAS

En un *ciclo* podrán recibirse grandes cantidades de *muestras*, pero sólo se almacenaran aquellas tres que pasen por un proceso de filtrado. El tiempo es una de las variables más importantes para lograr resultados estadísticos aceptables. No sólo se hace referencia a la importancia de registrar temporalmente cada *muestra*, sino que su recolección se haga durante un lapso de tiempo prolongado para permitir la interacción de los nodos/ganado en el campo.

De esta manera, filtrar será de gran utilidad para evitar una rápida saturación de la memoria con *muestras* sin relevancia para el estudio. El criterio general de filtrado será conservar las 3 *muestras* recibidas cuyo valor de *RSSI* sea el más próximo al valor patrón que existe en el amamantamiento del animal bajo estudio.

5.4.MÉTODOS DE GESTIÓN DE MEMORIA

¿Cuántas *muestras* podrán almacenarse en la memoria disponible? ¿Cuántos *ciclos* podrán registrarse? Como el tamaño de la Memoria de Información varía de acuerdo al microcontrolador, en esta sección se calculará la autonomía de la memoria usando al segmento de 512 bytes como tamaño relativo de memoria. De esta manera, el análisis será válido para cualquier microcontrolador de la familia MSP430F22XX ya que al multiplicar el valor calculado por la cantidad de segmentos, se obtendrá el valor total.

Para simplificar el cálculo de la autonomía relativa de la memoria, en cada *ciclo* donde se reciban *muestras*, se supondrá un total de tres, la máxima cantidad permitida por *ciclo* de acuerdo al criterio de filtrado adoptado. Así, con 512 bytes por segmento en memoria, dividiendo por 9 bytes que ocupan 3 *muestras*, obtenemos 56,88 *ciclos* por segmento o el equivalente de 170,66 *muestras* por segmento. Estas cantidades son sólo una referencia ya que es necesario el empleo de bytes o bits adicionales para el funcionamiento del firmware de gestión, sin los cuales el microcontrolador no contaría con criterios ni la capacidad para almacenarlas. Los algoritmos de almacenamiento y de lectura deben identificar el inicio y fin de cada *ciclo*, reconocer una *muestra* de un *timestamp* y de un *ciclo vacío*, y ubicar el *byte libre* a partir del cual podrá almacenarse en el siguiente *ciclo*.

Fue materia de estudio proponer y evaluar varios métodos de gestión para optimizar el uso de la memoria y poder almacenar durante la mayor cantidad de *ciclos* posibles la máxima cantidad de *muestras*. En otras palabras, el método elegido será aquel que permita acercarnos a los valores de referencia calculados al inicio del apartado (56,88 *ciclos* y 170,66

muestras por segmento). Se propusieron tres métodos cuya eficiencia fue contrastada en tres escenarios posibles que permiten reflejar cuantitativamente la autonomía relativa de la memoria en *ciclos* o *muestras* por segmento.

1° Escenario: 3 *muestras* por *ciclo*. En este escenario se evaluará la cantidad total de *muestras* por segmento que la memoria podrá almacenar, incluyendo los bytes o bits extras que el método en cuestión necesitará para funcionar.

2° Escenario: Un *ciclo* de 3 *muestras* seguido de un *ciclo sin muestras*. Aquí podrá cuantificarse la autonomía incluyendo, además, los bits y bytes necesarios para la gestión de los *ciclos vacíos*.

3° Escenario: Un *ciclo* de 3 *muestras* seguido de 3 *ciclos sin muestras*. Considerando que la aplicación consiste en una red de sensores dispersa en una gran superficie, los *ciclos sin muestras* podrían ser recurrentes. Este escenario cuantificará la autonomía contemplando la gestión de múltiples *ciclos vacíos*.

Las fórmulas usadas serán las siguientes:

$$\textbf{\textit{Cantidad de Ciclos relativa}}: \frac{Ciclos\ de\ Autonomía\ por\ segmento\ del\ método}{56{,}88}\ x\ 100$$

$$\textbf{\textit{Cantidad de Muestras relativa}}: \frac{Máxima\ Cantidad\ de\ Muestras\ por\ segmento\ del\ método}{170{,}66} x100$$

Además, los algoritmos serán evaluados por su complejidad para implementarlos en lenguaje de programación. El análisis será cualitativo y basado en dos de las más importantes funcionalidades con las que deberá contar. Estas consisten en la búsqueda del *byte libre* y en la búsqueda y devolución de cualquier número de *muestra* que se le solicite.

4.5.1.MÉTODO 1

Consiste en almacenar en cada *ciclo*, junto con las *muestras* recolectadas, un *timestamp* y una cabecera para indicar el inicio de un *ciclo*. El ordenamiento de la memoria se ilustra en la figura 10.

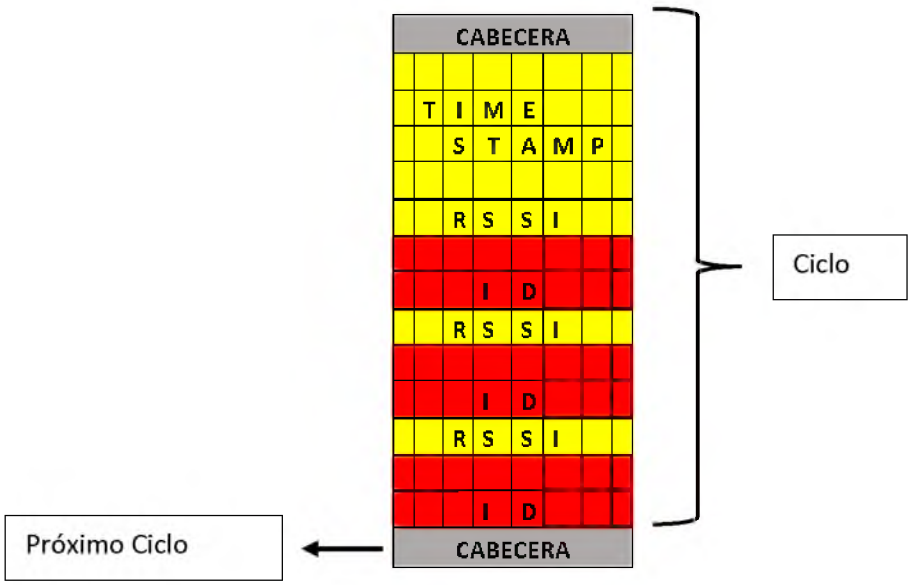

Figura 10: Representación gráfica del método 1.

El algoritmo de gestión sólo deberá saber las longitudes de los tres elementos (*Muestra*: 3 bytes, *timestamp*: 4 bytes, Cabecera: 1 byte) y el orden planteado. La cabecera, que consiste en 1 byte, indicará la cantidad de *muestras* a continuación del *timestamp*. Su valor nunca excederá el número 3 por el criterio de filtrado adoptado.

Calculemos para cada escenario:

1° Escenario: Si disponemos de 512 bytes por segmento, dividido entre 14 bytes (9 bytes de *muestras* + 4 bytes de *timestamp* + 1 byte de cabecera) dará una autonomía de 36,57 *ciclos* o el equivalente a 109,71 *muestras* por segmento. Así, al registrar el tiempo y usar una cabecera para brindar soporte al algoritmo, nos alejamos 20,30 *ciclos* de la autonomía ideal de 56,88 *ciclos* o 60,96 *muestras* de la de 170,66 *muestras*

Cantidad de Ciclos relativa: 64,29 %

Cantidad de Muestras relativa: 64,29 %

2° Escenario: Dado a que no es necesario registrar los *ciclos sin muestras*, este método daría una autonomía de 73,14 *ciclos* por segmento, el doble del 1° escenario. La cantidad de *muestras* que el método puede almacenar con el espacio disponible, es la misma del escenario 1.

Cantidad de Ciclos relativa: 128,58 %

Cantidad de Muestras relativa: 64,29 %

3° Escenario: Calculando de la misma manera, tendríamos 4 veces 36,57 =146,28 *ciclos* de autonomía. La cantidad de *muestras*, como en el escenario anterior, sigue siendo la misma.

Cantidad de Ciclos relativa: 257,17 %

Cantidad de Muestras relativa: 64,29 %

Ventajas del método:

- Los *ciclos sin muestras* no perjudican la autonomía de la memoria por mas que sean recurrentes. Esto se observa en el escenario N° 2 y en el N° 3 donde aumenta la cantidad de *ciclos* que pueden registrarse respecto al escenario N° 1, sin modificar la cantidad de *muestras* que pueden almacenarse.
- Los algoritmos de almacenamiento y lectura no requieren mucho procesamiento y son de fácil implementación.
  El inicio y fin de cada *ciclo* se identifica sólo con un byte de cabecera que, además, indica la cantidad de *muestras* que se han grabado a continuación del *timestamp*. Cada *ciclo* con *muestras* se registra con un orden establecido: byte de cabecera, *timestamp* y las *muestras*. Como los tamaños de estos tres elementos es conocido, la identificación del *byte libre* consiste en leer las cabeceras de cada *ciclo* hasta encontrar el byte 0XFF en lugar de una cabecera.
  La devolución de un número de *muestra* en particular consiste simplemente en acumular los valores de cada cabecera hasta coincidir o excederse del número buscado. A continuación, se apunta directamente a la dirección de memoria de la *muestra* y la lectura es inmediata

Desventajas del método:

- La autonomía relativa de *muestras* se ve notablemente disminuida en los tres escenarios analizados, respecto a los valores de referencia calculados al inicio de esta sección. Esto se debe a que el *timestamp* debe almacenarse en todos los *ciclos* con *muestras* para no perderse el rastro temporal luego de los *ciclos sin muestras*.

La importancia de registrar la mayor cantidad de *muestras*, nos lleva a proyectar un segundo método que supla esta deficiencia.

4.5.2.MÉTODO 2

Este método consiste en almacenar inicialmente un único *timestamp* y llevar el registro temporal dejando marcas al inicio de cada *muestra*, usando 1 bit.

Este bit no requiere espacio adicional dado a que podemos prescindir del bit más significativo del *RSSI*. Con este se representa un espectro de señales de *RSSI* de 0 a 63db de ganancia que nunca recibiremos. El motivo es que estas señales están afectadas por las pérdidas de dispersión dada la distancia que separa a un nodo receptor de uno transmisor y por ende, nunca serán amplificadas.

Este bit será una marca de *ciclo* que encabezará a todas las *muestras* y alternará de 1 a 0 y de 0 a 1 con cada cambio de *ciclo*. La imagen N° 11 ilustra lo mencionado.

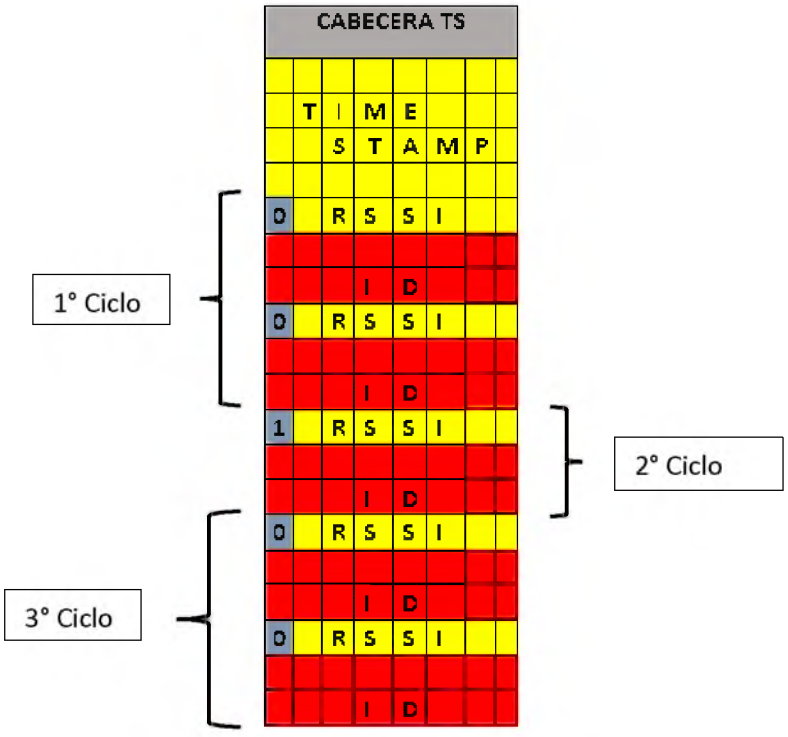

Figura 11: Representación gráfica del método 2.

Si por algún motivo el microcontrolador del nodo se reinicia, la referencia temporal se perderá. Para lidiar con esta posibilidad, el método contempla volver a grabar el *timestamp* en el primer *byte libre* seguido de lo almacenado anteriormente. Es por ello que es necesario registrarlo con un byte de cabecera, para que el algoritmo que ejecute este método lo reconozca y no sea malinterpretado como una *muestra*. Esta cabecera deberá contener un número cuyo uso será restringido para los demás elementos.

En este método será necesario registrar los *ciclos sin muestras* para no perder el rastro temporal. Para ello se hace uso de 1 byte encabezado con una marca de *ciclo*, seguido de todos sus elementos en cero. En la figura N° 12 se observa el almacenamiento de *ciclos* sucesivos en memoria.

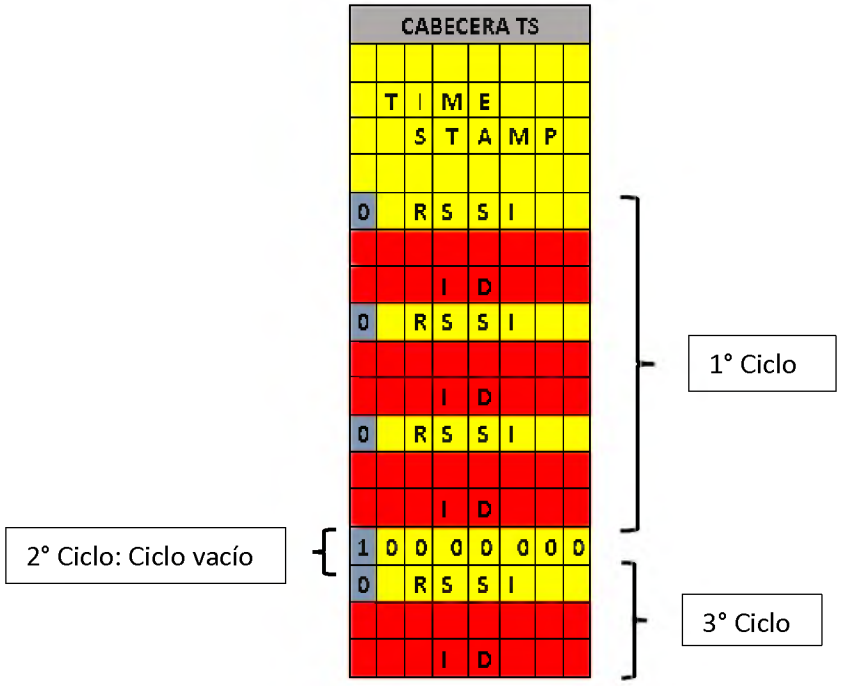

Figura 12: Representación gráfica del método 2.

A continuación se cuantificará la autonomía que el uso de este método proporciona. El *timestamp* se registra sólo al inicio de la memoria, pero para adoptar una postura más conservadora que contemple reiterados reseteos del microcontrolador y para simplificar los cálculos, se evaluará como si se grabara un *timestamp* al inicio de cada segmento. De esta manera, cada segmento contará con 507 bytes efectivos (512 menos 4 bytes de *timestamp* y menos 1 byte de cabecera) para el almacenamiento de *muestras* y el registro de *ciclos vacíos*.

1° Escenario: 507 bytes dividido entre 9 bytes de *muestras* dará una autonomía de 56,33 *ciclos* con 3 *muestras*. De esta manera, se podrán almacenar 169 *muestras*.

Cantidad de Ciclos relativa: 99,03 %

Cantidad de Muestras relativa: 99,02 %

2° Escenario: Por cada dos *ciclos* se usarán 10 bytes (9 bytes de *muestras* y 1 de registro de *ciclo vacío*). Así, se podrán almacenar 50,7 paquetes de 3 *muestras* y 1 byte de *ciclo vacío*. Esto equivale a una autonomía de 152,1 *muestras* (50,7 x 3 *muestras*) o 101,4 *ciclos* (50,7 x 2 *ciclos*, un *ciclo* con *muestras* y otro sin *muestras*).

Cantidad de Ciclos relativa: 178,27 %

Cantidad de Muestras relativa: 89,12 %

3° Escenario: Por cada 4 *ciclos* se requeriran 12 bytes (9 para 3 *muestras*, y 1 por cada *ciclo* sin *muestras*). Así, se podrán almacenar 42,25 paquetes de 3 *muestras* y 3 bytes de *ciclos vacíos*. Esto equivale a 126,75 *muestras* (42,25 x 3 *muestras*) o 169 *ciclos* (42,25 x 4 *ciclos*, un *ciclo* con *muestras* y tres sin *muestras*).

Cantidad de Ciclos relativa: 297,11 %

Cantidad de Muestras relativa: 74,27 %

Ventajas:

- Como no es necesario almacenar un *timestamp* por cada *ciclo*, aumenta la cantidad de *muestras* relativas entre un 74 y 99 %. Estos índices indican que más bytes son usados para almacenar *muestras* en contraste al método 1.
- Por el mismo motivo, la cantidad de *ciclos* relativa aumenta considerablemente respecto a la del método 1, permitiendo entre un 99 y un 297% de 56,88 *ciclos* ideales. Esto significa que ante una posible dispersión prolongada del ganado, que genera reiterados *ciclos vacíos*, la autonomía no se verá afectada y la memoria estará disponible para almacenar *muestras* cuando la dispersión disminuya.
- Recicla un bit de cada *muestra* para gestionar la memoria, sin pérdida de información.

Desventajas:

- Se observa una tendencia negativa en la cantidad de *muestras* que se pueden almacenar, respecto del valor de referencia ideal, ante reiterados *ciclos vacíos*. Hubo una reducción del 25% de autonomía relativa de *muestras* en el escenario 3 con sólo 3 *ciclos vacíos*. Esto se debe a que se destina 1 byte para el registro de los mismos.
- Es necesario implementar un algoritmo más inteligente capaz de identificar una *muestra* de un byte de registro de *ciclos vacíos*, identificar un cambio de *ciclo* de acuerdo a la alternancia de la marca y actualizarse sobre la marcha para no perder el registro temporal, desordenar el contenido o pisar información ya almacenada, durante una escritura en memoria

Esta debilidad del método por los *ciclos vacíos* es muy perjudicial para la aplicación de Seguimiento Genético de Ganado, donde se prevé una gran dispersión del ganado (y de los nodos) en el campo y en consecuencia una abundancia de *ciclos sin muestras*. Así, fue necesario pensar en un tercer método que haga del segundo uno que no desperdicie tanta memoria en *ciclos vacíos*.

4.5.3.MÉTODO 3

Este método consiste en extraer un segundo bit de la *muestra* para propósitos de gestión. Del número *ID* no puede extraerse ningún bit sin perderse capacidad de representación y limitar la aplicación. Así, la única alternativa restante, sin aumentar el tamaño de una *muestra*, es extraerlo del *RSSI* en detrimento de su resolución.

Dado a que no tener pérdidas (0 db) es imposible ya que nunca habrá una cercanía infinitesimal entre dos nodos y cómo las señales obtenidas de trasmisores muy lejanos (-64db) no aportarían información útil al análisis estadístico, se usaron sólo 6 bits delimitando una ventana de sólo 64 valores de *RSSI* entre 0 y -64db. Se pierde capacidad de representación pero es un filtrado inicial de *muestras* carentes de significado para la aplicación. De esta manera, podemos afirmar que no se perderá información relevante.

De esta forma tendremos dos bits de cabecera por cada *muestra* o byte de registro de *ciclos vacíos*:

- **Marca:** Primer bit para llevar un rastro temporal y están asociadas a un *timestamp* inicial que se almacena ni bien el microcontrolador es encendido o reseteado. Consiste en 1 bit que alterna de 0 a 1 por cada nuevo *ciclo*, permitiendo su identificación. Así, si cada *ciclo* se ejecuta cada un periodo de

tiempo determinado, se puede calcular el horario y fecha de cualquier *ciclo* sumándole, al *timestamp* asociado, la cantidad de transiciones de 1 a 0 y de 0 a 1 multiplicado por la cantidad de tiempo que trascurre entre *ciclos*.

- **Tipo de información:** Este bit es el segundo e indica si los bytes a continuación corresponden a una *muestra* o a un byte de registro de *ciclos vacíos*. Para el caso de una *muestra*, se usó el 1 lógico y para el registro de un *ciclo vacío*, un 0 lógico.

La importancia de este segundo bit radica en que permite aprovechar el byte de registro de *ciclos vacíos* para cuando sean recurrentes. Luego de los bits de cabecera, los 6 bits restantes se usarán para registrar *ciclos vacíos* consecutivos. Permanecerán en 1 si no los hubiera o irán volviéndose 0 a medida que suceden. Hasta 7 *ciclos vacíos* sucesivos podrán registrarse sin necesidad de ocupar el siguiente byte.

En la imagen N° 13 podrá apreciarse varios ejemplos de bytes para registro de *ciclos vacíos*. La x indica que el bit en cuestión puede adoptar indistintamente un 1 o un 0 lógico.

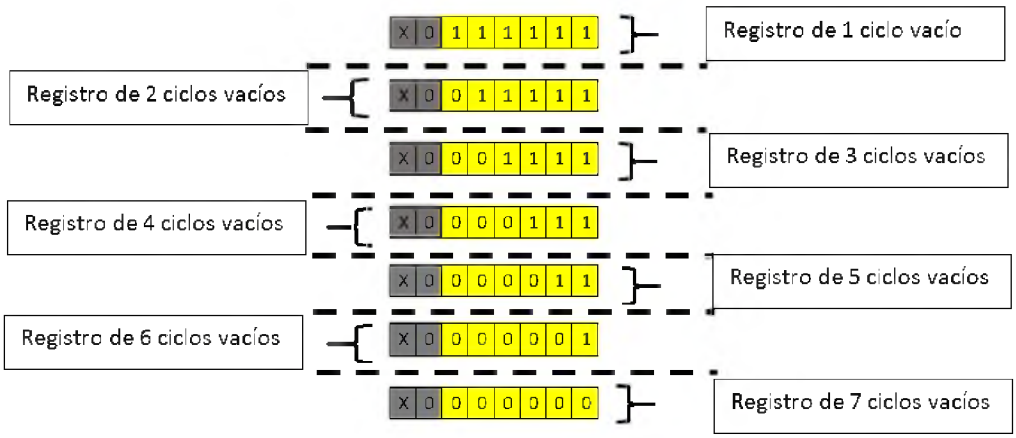

Figura 13: Representación gráfica del registro de ciclos vacíos en el método 3.

Al igual que en el método 2, todos los *timestamp* almacenados, ya sea al inicio como en cualquier momento después de un reseteo del microcontrolador, deberán tener una cabecera para que los algoritmos de gestión pueden diferenciarlos de una *muestra* y de un byte de registro de *ciclos vacíos*. Se eligió la cabecera 0x7E (0b 0111 1110) dado a que nunca se igualará a un byte de registro de *ciclos vacíos*. A pesar de esto, podría igualarse al 1° byte de una *muestra* en un *ciclo* con marca igual a 0. Por ende, el algoritmo deberá contemplar esta restricción y evitar que se registre un *RSSI* equivalente a 0x3E (0b 11 1110). No será

necesario tomar recaudos en los bytes intermedios de cada *muestra* o *timestamp* debido a que el algoritmo nunca accederá a leerlos para su funcionamiento.

En la figura N° 14 se muestra el método en cuestión.

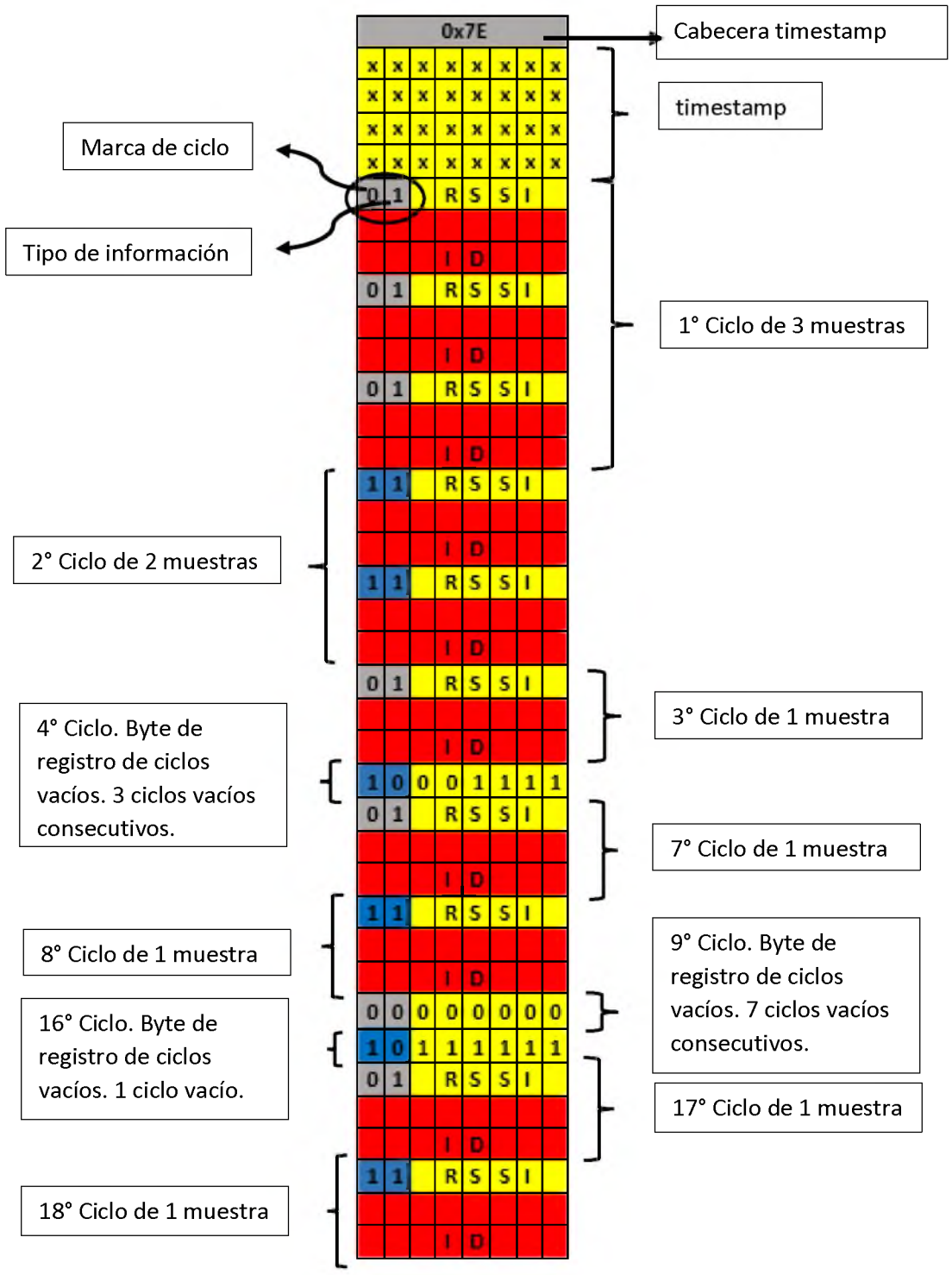

Figura 14: Representación gráfica del método 3.

Finalmente, se calcula la autonomía relativa del método.

1° Escenario: 507 bytes dividivo entre 9 bytes (3 *muestras* de 3 bytes) da una autonomía igual a la del método 2: 56,33 *ciclos* o 169 *muestras*.

Cantidad de Ciclos relativa: 99,03 %

Cantidad de Muestras relativa: 99,02 %

2° Escenario: El cálculo es igual al método 2, por lo que dará una autonomía de 152,1 *muestras* o 101,4 *ciclos*.

Cantidad de Ciclos relativa: 178,27 %

Cantidad de Muestras relativa: 89,12 %

3° Escenario: Por cada 4 *ciclos* se requeriran 10 bytes (9 para 3 *muestras*, y sólo 1 por los 3 *ciclos sin muestras*). Así, se podrán almacenar 50,7 paquetes de 3 *muestras* y 1 byte de 3 *ciclos vacíos*. Esto equivale a 152,1 *muestras* (50,7 x 3 *muestras*) como en el escenario 2 o 202,8 *ciclos* (50,7 x 4 *ciclos*, un *ciclo* con *muestras* y tres sin *muestras*).

Cantidad de Ciclos relativa: 356,54 %

Cantidad de Muestras relativa: 89,12 %

Estos porcentajes se mantendrán con hasta 7 *ciclos vacíos* consecutivos.

Ventajas:

- Al no ser necesario almacenar un *timestamp* y una cabecera por cada *ciclo* con *muestras*, aumenta la cantidad de *muestras* que pueden almacenarse entre un 89 y 100 %, respecto a los valores de referencia ideales calculados al inico de la sección. Estos índices indican que más bytes son usados para almacenar *muestras* en contraste al método 1 y al método 2.
- Recicla dos bits de cada *muestra* para gestionar la memoria, evitando la necesidad de usar 1 byte de cabecera como el método 1 y sin pérdida de información.
- La cantidad de *ciclos* relativa aumenta considerablemente respecto a las del método 1 y 2, permitiendo entre un 99% y un 356,54% de 56,88 *ciclos* ideales. Esto significa que ante una posible dispersión prolongada del ganado, que genera reiterados *ciclos*

vacíos, la autonomía no se verá afectada y la memoria estará disponible para almacenar *muestras* cuando la dispersión disminuya.

- La recurrencia de *ciclos vacíos* no impacta tan negativamente en la autonomía de la memoria como en el método 2, al haber transformado el byte de registro de *ciclos vacíos* en un acumulador de hasta 7 *ciclos vacíos* consecutivos.

Desventajas:

- Es necesario implementar un algoritmo aún más inteligente que el método 2, que además tenga la capacidad de interpretar dos bits de cabecera y que relacione el cuerpo del byte de registro de *ciclos vacíos* como *ciclos* independientes para poder calcular así la cantidad de *ciclos* trasncurridos.

En la tabla N° 1 podemos observar comparativamente el desempeño de los tres métodos en los tres escenarios propuestos. En ella se observa la conveniencia del método N° 3 al ser el que mayor cantidad de *muestras* almacena, entre un 89% a un 99% respecto al valor de referencia ideal de 170,66 *muestras*. Estos porcentajes, además, reflejan la capacidad de método de sobrellevar recurrentes *ciclos vacíos* sin sacrificar autonomía de memoria. Finalmente, gracias a la gestión eficiente del método 3, se puede extender la cantidad de *ciclos* de la memoria, y en consecuencia la duración del estudio, entre un 99% y un 356% respecto al valor de referencia ideal de 56,88 *ciclos* en caso de una prolongada dispersión de los nodos de la red. De esta manera, los algoritmos de gestión realizados fueron diseñados en base al método N° 3.

Tabla 1: Comparación de los tres métodos de gestión analizados.

		ESCENARIO					
		1		2		3	
		CICLOS (%)	MUESTRAS (%)	CICLOS (%)	MUESTRAS (%)	CICLOS (%)	MUESTRAS (%)
MÉTODO	1	64,29	64,29	128,58	64,29	257,17	64,29
	2	99,03	99,02	178,27	89,12	297,11	74,27
	3	99,03	99,02	178,27	89,12	356,54	89,12

5.5.LIBRERÍA PARA LA GESTIÓN DE MEMORIA

La librería de gestión de *muestras* y *parámetros* se llamó Sample.c y cuenta con múltiples algoritmos para administrar la memoria como el método 3 propone en la anterior sección. Estos son:

Flash_init(): este algoritmo cumple la función de actualizar el estado actual de la Memoria Principal destinada al almacenamiento de *muestras*. Se debe ejecutar sólo cuando se enciende el microcontrolador para orientar a los demás servicios de la librería en el uso de la memoria. Este servicio procede ejecutando lecturas sucesivas desde el primer byte de la memoria, identificando cada cabecera, hasta encontrar el *byte libre* para la siguiente escritura.

SAMPLE_write_time_stamp(): este servicio deberá ejecutarse al inicio del microcontrolador para registrar el primer *timestamp* al cual estarán asociado las *muestras* o *ciclos vacíos* subsiguientes. Luego de cualquier reseteo, también deberá ejecutarse, para así registrar el *timestamp* a partir del cual el microcontrolador restableció su funcionamiento y no perder el rastro temporal.

SAMPLE_new_sample (uint16_t ID, uint8_t RSSI): esta función almacena temporalmente el *ID* y su *RSSI* asociado en una tabla dinámica ubicada en RAM. Puede ejecutarse reiteradas veces y sólo se preservarán las tres combinaciones cuyo *RSSI* se aproxime al *RSSI* patrón de amamantamiento. Esta es la función que se encarga del filtrado de *muestras*.

SAMPLE_save_cycle(): Este servicio almacenará todo el contenido de la tabla dinámica en la memoria flash, de forma permanente y en formato de *muestra*, con los encabezados correspondientes. En caso de no haber contenido en la tabla, se procederá a almacenar un *ciclo vacío* con el encabezado adecuado. Cualquier sea el caso, esta actualizará el estado de la memoria para las futuras grabaciones. Si no hay espacio en flash, no se almacenará nada.

SAMPLE_new_cycle(): Borra todo contenido de la tabla, preparándola para el siguiente *ciclo*.

Uint8_t SAMPLE_memory_status (uint16_t* number_cycles, uint16_t* number_of_samples, uint16_t* number_empty_cycle, uint16_t* number_time_stamps): Esta función devuelve si hay o no disponibilidad en la memoria. Además, a través de punteros, devuelve la cantidad de *ciclos*, de *muestras*, de *ciclos vacíos* y de *timestamps* a lo largo de toda la memoria.

Uint8_t SAMPLE_read_samples(uint16_t number_request, uint8_t* RSSI, uint16_t* ID, uint32_t* time_stamp): devuelve si el número de *muestra* está disponible y, en caso de estarlo, la *muestra* solicitada a través de punteros y descompuesta en *RSSI*, *ID*, *timestamp* asociado y los *ciclos* transcurridos desde este último, para el cálculo temporal.

SAMPLE_erase_sample_segment (): algoritmo que borra todos los segmentos de la Memoria Principal empleada para el almacenamiento de *muestras*.

Para la Gestión de *parámetros*, los servicios son los siguientes:

uint8_t SAMPLE_write_parameter_byte (uint8_t byte, uint16_t segment_position): Este servicio almacena, en el segmento B de la memoria de información, un *parámetro* de la red WSN ingresado a través del argumento **byte** en la posición indicada a través de **segment_position** (de 0 a 63).

uint8_t SAMPLE_read_parameter_byte (uint8_t* parameter_byte, uint8_t segment_position): Esta función devuelve por puntero un *parámetro* de un byte ubicado en la posición 0 a 63 del segmento B, pasada como argumento.

SAMPLE_erase_parameter_segment (): Este servicio borra todo el segmento B usado para almacenamiento de *parámetros*.

En la siguiente sección analizaremos el funcionamiento en conjunto de estos servicios.

5.6.PROCESO GENERAL DE GESTIÓN DE MEMORIA

En el diagrama de flujo de la figura N° 15 se ilustra una recomendación de uso de la librería Sample.c. Los recuadros en azul es donde la librería acciona y en amarillo donde la

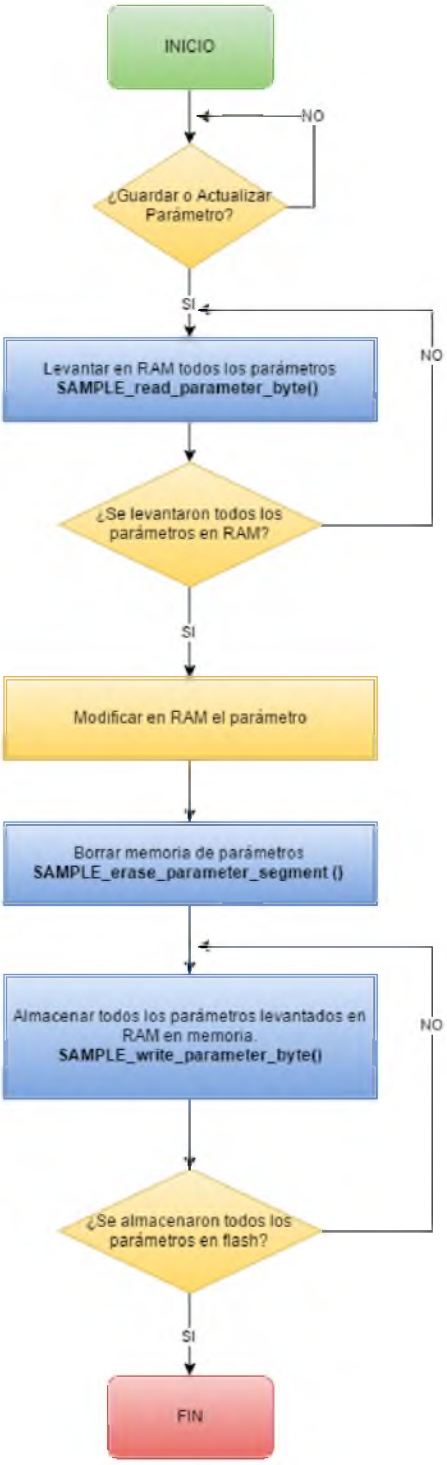

Figura 15: Diagrama de flujo recomendado para el uso de los servicios de Sample.c para el almacenamiento de parámetros.

aplicación estará presente y podrá actuar libremente. En caso de llenarse la memoria, sólo tendrán efectos los servicios de lectura y de borrado de memoria.

El proceso para almacenamiento de *parámetros* es el que se muestra en el diagrama de flujo de la imagen N° 16. Este proceso consiste en levantar, en memoria RAM, todos los bytes usados del segmento B para no alterar la información ya almacenada. El motivo radica en que no se puede sobrescribir una celda de memoria que tiene un 0 lógico almacenado. Sólo a través de la ejecución de un borrado de a segmento se podría llevarla a un 1 lógico.

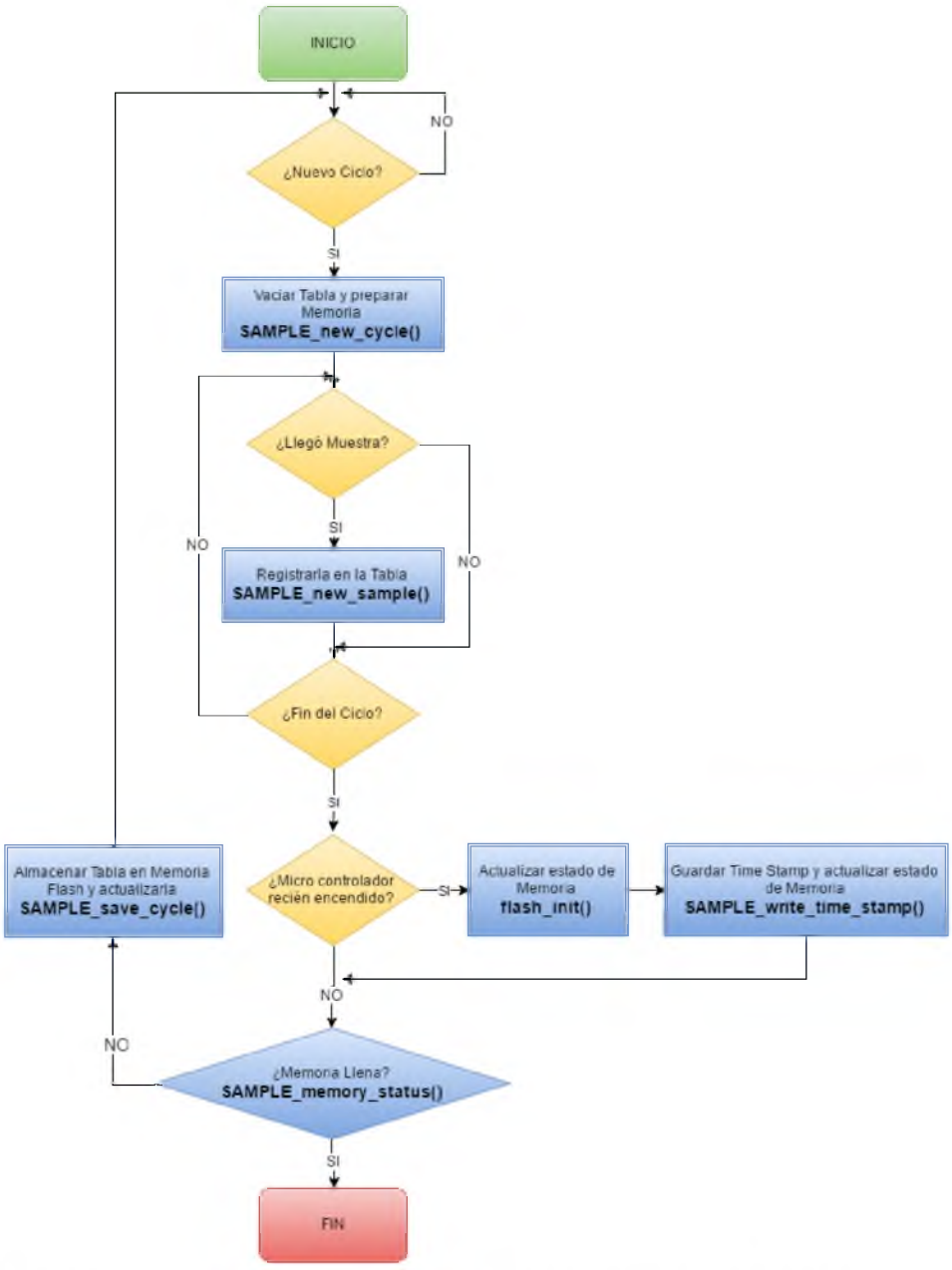

Figura 16: Diagrama de flujo recomendado para el uso de los servicios de Sample.c durante el almacenamiento de muestras.

5.7.¿POR QUÉ NO ANEXAR MÁS MEMORIA A CADA NODO DE LA WSN?

Cada escritura en flash tiene una penalización energética que impacta directamente en la duración de la batería del nodo. Ésta, junto con el funcionamiento de los circuitos de RF, son las actividades de mayor consumo. Así, para cumplir con el mandamiento de bajo consumo de una WSN, se debe prolongar la vida de la red optimizando los recursos disponibles lo máximo posible.

A modo de comparación, el sub-sistema de procesamiento comandado por el microcontrolador MSP430F2254 en modo activo consume 270uA (ultra bajo consumo), mientras que una escritura en flash conlleva un consumo de 5mA durante 90u segundos por palabra (2 bytes), unas 18,5 veces más.

Más memoria flash conlleva un potencial aumento de accesos a escritura proporcional al tamaño incorporado y, en consecuencia, un consumo energético mayor. Es por esto, que el enfoque de este trabajo final no sólo se orientó a gestionar memoria, sino que además buscó optimizar el uso de la escasamente disponible.

6.ALGORITMOS DE COMPRESIÓN

6.1.INTRODUCCIÓN

Dada la escases de memoria disponible y la restricción energética para agregar más memoria, se analizó la viabilidad de dotar a la red con la capacidad de compresión de datos a fin de liberar espacio y prolongar la recolección de información y la duración del estudio comportamental.

La compresión de datos es el arte o ciencia de representar información de una forma compacta. Estas representaciones compactas son creadas a partir de la identificación y el uso de estructuras que existen en los datos a comprimir o de las características del usuario que va a usar esos datos. Estos pueden ser caracteres en un documento de texto, números de muestras de señales de audio o de imagen o secuencias de números generados por otros procesos.

Aunque no esté explícito, cuando se habla de técnicas de compresión o algoritmos de compresión nos estamos refiriendo a dos algoritmos. El de compresión, que a partir de los datos de entrada genera una representación de ellos con menos bits, y el descompresor, que trabaja sobre la representación comprimida para generar una reconstrucción de los datos originales.

Esta reconstrucción puede ser exacta a los datos iniciales o no. De esta manera, la compresión puede dividirse en dos grandes clases: *Esquemas de compresión sin pérdida de información*, donde el descompresor reconstruye exactamente los datos originales y los *Esquemas de compresión con pérdida*, que permite una mayor compresión pero donde el descompresor no puede reconstruir exactamente los datos iniciales [10].

La elección del esquema adecuado depende de la aplicación. En el caso de videos, fotografías y sonidos, las pérdidas son admisibles si a simple vista o para el oído no son percibidas. Ahora bien, para texto, una pérdida de información puede costar el sentido del mismo. En el caso de este trabajo, no hay lugar para la pérdida de información ya que las *muestras* cuentan con un número *ID* cuyo truncamiento podría generar otro *ID* presente en la red, desviando el contenido de la *muestra* (*RSSI* y Fecha del evento) a otro nodo y en consecuencia, generándose conclusiones comportamentales incorrectas. Finalmente, una disminución adicional de la resolución del *RSSI* o la pérdida de la marca temporal, también generaría datos incorrectos.

Definido el uso de un *esquema de compresión sin pérdida*, debemos reconocer la existencia de una segunda clasificación. Aquellos que usan una Metodología Estadística, donde el algoritmo compresor necesita conocer la distribución de probabilidades de los elementos de la fuente de antemano, que si no es posible, leer toda la fuente y calcularla se vuelve una tarea primaria, haciendo la compresión un proceso de dos etapas. Por otro lado, están los que usan una Metodología en Línea, donde el algoritmo va comprimiendo a medida que va leyendo la información de la fuente, haciendo al proceso de compresión de una sola etapa [11].

Se estudió los algoritmos en los que se basan los compresores más populares. Se analizó su conveniencia de acuerdo al contexto de este proyecto, se eligió uno, se implementó en lenguaje C, se crearon varias muestras con datos de entrada posibles, se comprimieron y se calculó la relación entre los byte de entrada y los byte de salida (Tasa de compresión).

En la siguiente sección, se estudiarán 2 algoritmos que siguen una *metodología de compresión sin pérdida* debido a su empleo en compresores conocidos como el RAR, el GZIP, el BZIP2, el ZIP, el 7Z, entre otros.

6.2.ALGORITMOS DE COMPRESIÓN SIN PÉRDIDA DE INFORMACIÓN

2.6.1.ALGORITMO BASADO EN LA CODIFICACIÓN HUFFMAN

Los códigos generados con este tipo de codificación se llaman códigos de Huffman y son códigos prefijos, es decir, ninguna palabra del código es prefijo de otra. De esta manera, un decodificador, al reconstruirlos, no enfrentará ambigüedades ya que hay un solo resultado posible.

Los códigos de Huffman codifican aquellos símbolos que aparecen con mayor frecuencia (mayor probabilidad de ocurrencia) con una palabra de pocos bits comparado con aquellos de menor frecuencia [10]. Una relación inversa que permite a simple vista descubrir cómo se produce una reducción de tamaño. Esta forma de codificar permite comprimir aquellos datos de entrada cuyos símbolos no presenten igual probabilidad de ocurrencia. Si ese no fuera el caso, simplemente se llegaría a codificar como el sistema binario tradicional y no habría compresión.

El armado de un código de Huffman puede apreciarse con el siguiente ejemplo:

El texto inicial o datos de entrada será la palabra MISSISSIPPI en lenguaje ASCII. Es un proceso de dos pasos, en donde el algoritmo de compresión deberá leer los datos de entrada dos veces. La primera para identificar los símbolos presentes y medir su frecuencia de aparición y la segunda para codificar los símbolos de entrada, en base a esta. Es evidente que la palabra Mississippi no contiene todos los símbolos del alfabeto latino, lo cual ayuda a que la codificación no requiera de tantos bits y la relación de compresión sea aún mejor.

Para cumplimentar con el primer paso, se genera una tabla con cada símbolo presente en el texto a codificar, y se le asigna su frecuencia de aparición, siendo 11 el número total de letras. La palabra P denota probabilidad.

M = 1 (P = 1/11)

I = 4 (P = 4/11)

S = 4 (P = 4/11)

P = 2 (P = 2/11)

Una vez creada la tabla, el armado del código puede graficarse con un diagrama de árbol binario, donde los símbolos serán los nodos hoja de la base:

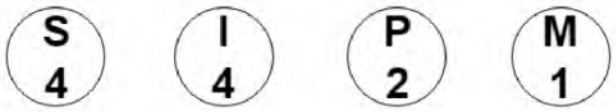

Figura 17: Base de un Diagrama binario. Cada nodo contiene un símbolo y su frecuencia de aparición en los datos de entrada.

A continuación, se tienen que ir formando nodos parientes a partir de la sumatoria de la frecuencia de los dos nodos con menor probabilidad, hasta llegar al nodo raíz. Paso a paso se muestra lo mencionado:

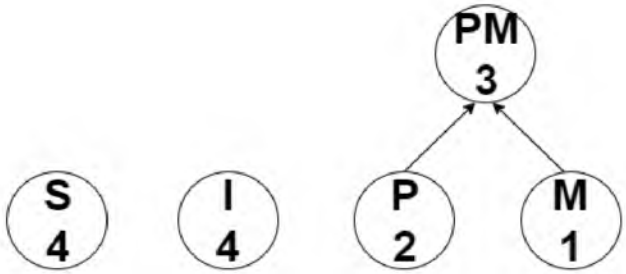

Figura 18: Ejemplo de Codificación Huffman.

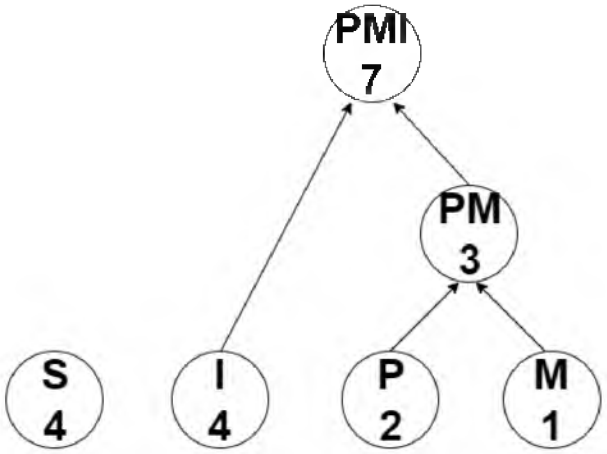

Figura 19: Ejemplo de Codificación Huffman.

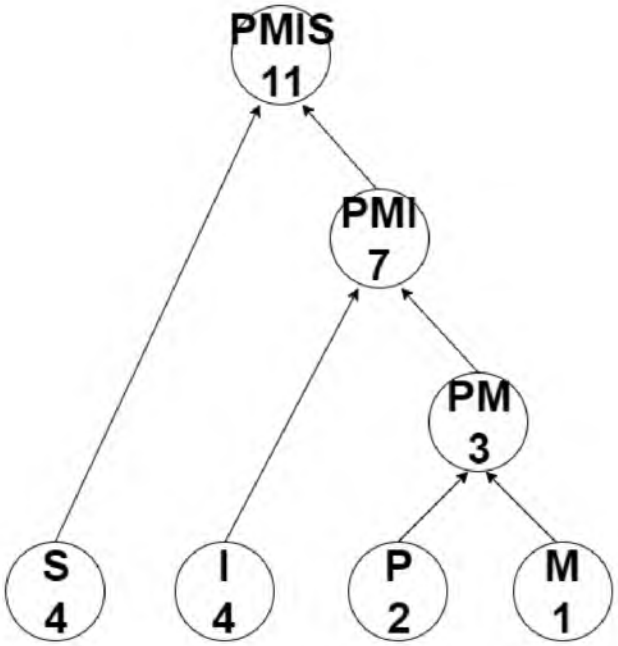

Figura 20: Ejemplo de Codificación Huffman.

Finalmente, a las ramas izquierdas se le asigna un 1 y a las derechas un 0 y la lectura desde arriba hacia los nodos de la base, darán la codificación de cada símbolo. La figura N° 21 muestra lo mencionado.

Así, los símbolos quedan codificados de la siguiente manera:

S = 1

I = 01

P = 001

M = 000

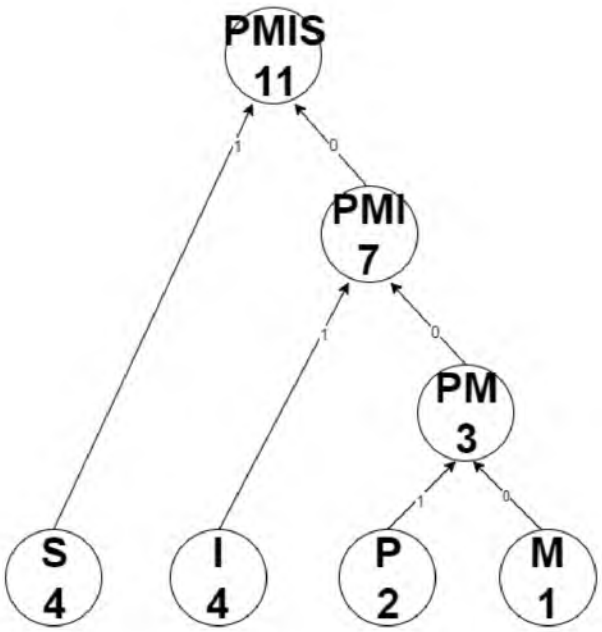

Figura 21: Árbol binario de Huffman.

Para finalizar la compresión, deberá releerse los datos de entrada y codificarlos. Mississippi quedaría codificado de la siguiente manera:

000 01 1 1 01 1 1 01 001 001 01

Un total de 21 bits. Si Mississippi estuviera codificado en ASCII, 88 bits hubieran sido necesarios. Así, la relación de compresión es de un 25% aproximadamente. Si se hace el ejercicio de descomprimirlo podrá descubrirse que no surgen ambigüedades a la hora de asignar los símbolos, al ser un código prefijo.

Ahora bien, si tuviéramos solamente el siguiente texto de entrada en ASCII: ABCDEFGH, donde la frecuencia de ocurrencia es igual para todos los caracteres, la tabla de codificación sería la siguiente:

A = (P=1/8) 000

B = (P = 1/8) 001

C = (P = 1/8) 010

D = (P = 1/8) 011

E = (P=1/8) 100

F = (P=1/8) 101

G = (P=1/8) 110

H = (P=1/8) 111

Así, se obtuvo la conocida codificación binaria empleada en los sistemas digitales, donde cada símbolo usa 3 bits debido a que son 8 elementos. De esta forma, sólo habría compresión por la ausencia de todos los símbolos del alfabeto latino, lo cual por lo general nunca ocurre en los datos de entrada de texto. Es por eso, que la distribución de probabilidades del texto de entrada es de suma importancia para la eficiencia del compresor de Huffman. Además, la selección de la unidad mínima que se considerará como símbolo a codificar, debe ser cuidadosamente elegida. En un texto extenso, elegir como símbolo el carácter individual, sería contraproducente para el codificador, debido a que la distribución de probabilidad sería una campana de Gauss plana. La palabra en ASCII de 1 byte, sería mucho más conveniente dado a que tendrá una distribución más normal (campana de Gauss delgada). En otras palabras, la Entropía de Shannon debe ser considerada. La misma indica la cantidad de información promedio que contienen los símbolos de la fuente. Aquellos con menos probabilidad aportarán más información que aquellos con una mayor. Cuando tenemos una distribución pareja de esta probabilidad en los distintos símbolos, significa que todos aportan información relevante y en este caso, la Entropía es máxima y el codificador de Huffman no sería eficiente. En caso de ser empleada esta codificación, la Entropía de la fuente deberá ser calculada como primera medida.

Debe mencionarse que para la descompresión, el compresor debe almacenar la tabla de ocurrencia de los símbolos para pasársela al descompresor y que pueda interpretar y reconstruir los datos comprimidos. Contextualizando este algoritmo para el uso en una red WSN aplicada al Seguimiento Genético de Ganado, donde la premisa de usar la menor cantidad de recursos es ley, el uso de este algoritmo no es factible. Suponiendo que se eligiera cómo símbolo el byte, el peor de los casos, que no debería descuidarse, sería la existencia de 256 bytes diferentes. Esto implicaría que la tabla de ocurrencia, como mínimo, requeriría de 256 bytes. Este número equipara a la mitad de la memoria RAM disponible para el funcionamiento normal del nodo. Además, esta tabla tiene que guardarse en flash, por lo que la tasa de compresión del algoritmo debería ser muy favorable para justificar que su almacenamiento ocupe medio segmento de memoria de los escasamente disponibles para *muestras*. Finalmente, contemplar una doble lectura de la memoria flash durante la segunda etapa, donde se almacenan los datos de entrada para el algoritmo y donde se almacena la tabla, implicaría un gasto energético extra. De esta manera, se descarta este algoritmo y se pasa a explicar, en la siguiente sección, al próximo candidato.

2.6.2.ALGORITMO LZ77

El algoritmo LZ77 (de Abraham Lempel, Jacob Zivy 1977) basa su funcionamiento en un diccionario que es una porción de los datos de entrada ya codificados. El codificador examina una secuencia de los datos de entrada a través de una "*ventana deslizante*" (Sliding window). Esta consiste en dos partes, un "*buffer de búsqueda*" o "*diccionario*" (Search buffer), que contiene un porción de la secuencia codificada recientemente, y un "*buffer adelantado*" (Look-ahead buffer) que contiene la próxima porción de la secuencia a codificar. En la imagen número 22 se puede apreciar una *ventana deslizante* de un tamaño elegido para que a fines prácticos se pueda explicar su funcionamiento. Su tamaño, generalmente, ronda las decenas o centenas de bytes de acuerdo a la aplicación.

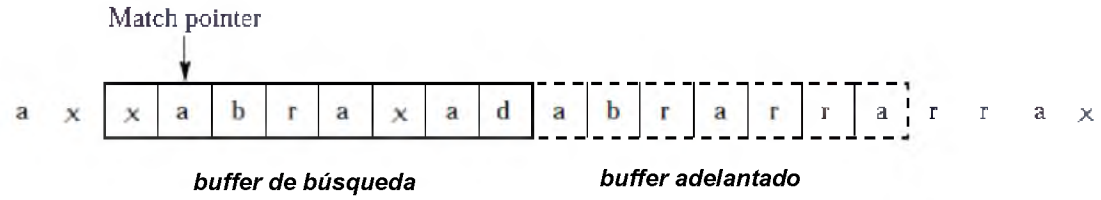

Figura 22: Ventana deslizante para el compresor LZ77.

Para codificar la secuencia presente en el *buffer adelantado*, el algoritmo mueve un puntero de derecha a izquierda desde la posición inicial del *buffer de búsqueda* hasta encontrar un carácter coincidente con el primero, de izquierda a derecha, del *buffer adelantado*. La distancia, desde el inicio de este buffer al puntero, se denomina "*offset*" (Off). A continuación, el codificador examina los símbolos siguientes a la ubicación del puntero (hacia la derecha) para revisar si hay más coincidencias con los símbolos consecutivos a la coincidencia ya encontrada. El número de símbolos consecutivos en el *buffer de búsqueda* que coinciden con los del *buffer adelantado*, empezando desde el primer símbolo, se llama "*longitud coincidente*" (Len). Una vez que el codificador encuentra la coincidencia más larga, lo codifica con una terna consistente en el *offset*, la *longitud coincidente* y el próximo carácter (Car) que sigue a la secuencia coincidente <0ff, Len, Car > [10].

La importancia de mandar el tercer elemento en la terna, radica en que es necesario contemplar la situación en la que no se encuentra una coincidencia de un símbolo del *buffer adelantado* en el *buffer de búsqueda*. En este caso, la terna se completa con un *offset* de cero, una *longitud coincidente* de 0 y el tercer elemento corresponde al valor en sí que no se pudo codificar.

Una vez encontrada y codificada la coincidencia en una terna, la *ventana deslizante* se desplaza Len + 1 hacia la derecha en búsqueda de una nueva porción de secuencia a codificar.

Hay tres posibles situaciones que pueden presentarse durante la codificación y son:

1. No hay coincidencia en el siguiente carácter a codificar.
2. Hay una coincidencia.
3. La coincidencia se extiende hacia adentro del b*uffer adelantado*.

Estas tres situaciones se analizarán con un ejemplo, suponiendo la siguiente secuencia a codificar ACCBADACCBACCBAGI y una *ventana deslizante* de 15 elementos (9 para el *buffer de búsqueda* y 6 para el *buffer adelantado*).

En la imagen N° 23 la flecha roja muestra el carácter del *buffer adelantado* que se buscará en el *buffer de búsqueda* y el cículo rojo el *offset* donde se produjo la coincidencia en el *buffer de búsqueda*. La linea verde se extenderá junto con los caracteres coincidentes, para remarcar la *longitud coincidente*.

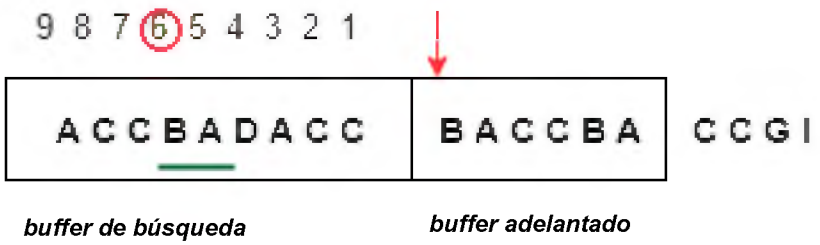

Figura 23: Búsqueda de coincidencia en la compresión LZ77.

Se emite la siguiente terna <6, 2, C>. Se desplaza 2 + 1 la *ventana deslizante*, y procedemos a codificar la siguiente secuencia que se muestra en la figura N° 24.

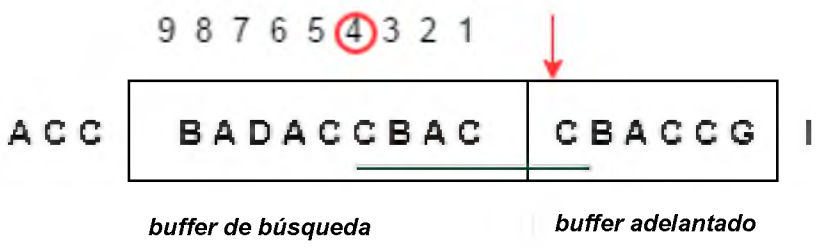

Figura 24: Coincidencia mayor que ingresa en el buffer de adelanto.

Así se obtiene la siguiente terna <4, 5, G>. Notar que se ingresó al *buffer de búsqueda* y no trajo consecuencias. Además notar que había dos posibles coincidencias en el *buffer de búsqueda*, y que el algoritmo optó por la de mayor longitud. Luego de la codificación, la *ventana deslizante* se desplaza 5 + 1 posiciones como en la imagen 25.

Figura 25: Ejemplo de cuando no se encuentra una coincidencia.

La terna resultante es <0, 0, I> ya que no hay coincidencias.

Si la longitud del *buffer de búsqueda* es S, la de la *ventana deslizante* (Sumatoria de *buffer de búsqueda* y *buffer adelantado*) es W, y la longitud del alfabeto de la fuente es A, la cantidad de bits necesarios para codificar la terna usando longitudes fijas es log2 (S) + logS (W) + log2 (A).

Este algoritmo asume que los patrones van a ocurrir cerca entre si ya que su *diccionario* consiste en la secuencia de caracteres inmediatamente predecesora de la presente en el *buffer adelantado*. Así, es importante la elección de la longitud del *diccionario* dado a que si la mayoría de las coincidencias caen por fuera del mismo, la tasa de compresión sería negativa, agrandando el archivo de salida comparado con el de entrada.

Llevar este algoritmo a una red WSN parece prudente. No requiere más que un buffer deslizante y el consumo de RAM estará dado por la longitud del mismo. Requiere una sola lectura de los datos de entrada y el algoritmo no conlleva un procesamiento complejo. De esta manera, se escogió implementar este algoritmo para analizar su tasa de compresión de acuerdo a datos de entrada modelados a lo que se espera que suceda con la red, en campo abierto.

6.3.IMPLEMENTACIÓN DEL ALGORITMO DE COMPRESIÓN SELECCIONADO

Los datos de entrada se generaron contemplando las siguientes suposiciones iniciales:

1. El 50% de los *ciclos* son vacíos y en el 50% de los *ciclos* restantes se reciben *muestras*.
2. Del 50% de *ciclos* con *muestras*, en el 90% se reciben tres *muestras* por *ciclo* de las cuales 2 pertenecen al nodo de la Madre.
3. Del 10% restante, se reciben tres *muestras* por *ciclo* pero ninguna pertenece al nodo de la Madre.
4. Todos los valores *RSSI* y los ID son generados aleatoriamente, con la premisa de estar contextualizados de acuerdo a la *muestra* a la que pertenezcan.
5. El orden de los *ciclos* también fue totalmente aleatorio.

Lo anterior se resume en el diagrama N° 26.

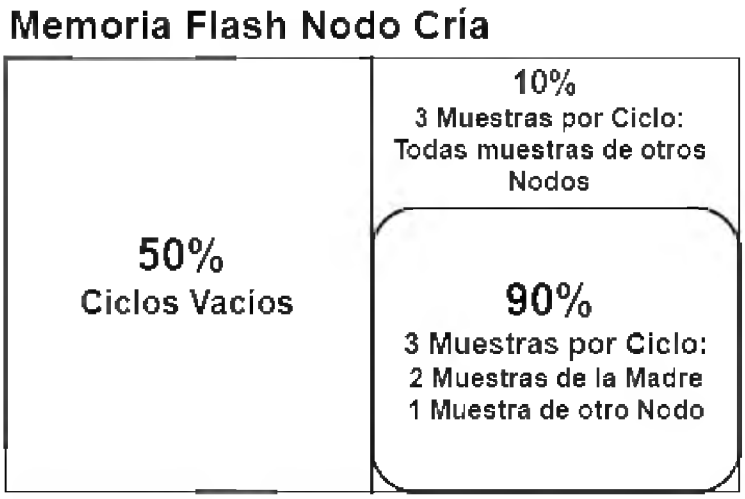

Figura 26: Fuente de entrada modelo para la compresión.

Se implementó el algoritmo en lenguaje C en la IDE Oracle NetBeans empleando una computadora para disponer de mayor flexibilidad de los recursos de memoria y de procesamiento disponibles y para que en esta etapa evaluativa se priorice el estudio de la tasa de compresión del algoritmo, sin ningún tipo de restricción de hardware. La adaptación del código a la red WSN sería una tarea menor pero posterior, y sólo si se obtienen resultados favorables.

Se implementó el algoritmo LZ77 con una *ventana deslizante* y una terna de salida configurables en diferentes tamaños para ir probando diferentes variantes.

El mejor resultado se obtuvo con la siguiente combinación: una *ventana deslizante* de 260 bytes donde 256 bytes fueron destinados al *buffer de búsqueda* y 4 bytes al *buffer*

adelantado. La unidad mínima que se eligió, de la fuente de entrada, fue el carácter hexadecimal de 0x0 a 0xF con lo cual se necesitaron destinar 4 bits para su representación en la terna de salida. Con 256 bytes, el *buffer de búsqueda* es capaz de almacenar 512 caracteres hexadecimales, necesitándose 9 bits para representar el *offset* en la terna de salida. Se acotó la longitud de las coincidencias encontradas a 8 caracteres dado a que la mayoría de los bits de la terna de salida se destinaron a aumentar el tamaño del *diccionario*, con la finalidad de incrementar la tasa de coincidencias más largas y en consecuencia la compresión. De esta manera, quedaron 3 bits disponibles para representar coincidencias de hasta 8 caracteres. Así, la terna de salida se conformó con 2 bytes.

La tasa de compresión dio negativa por unos cuantos bytes (120 bytes promedio) en reiterados intentos con variaciones aleatorias en los datos de entrada. El algoritmo requiere de 2 bytes para representar, con la terna de salida, desde 1 hasta 8 coincidencias. Los 2 bytes, si se dejara de lado la compresión, permitirían representar 4 caracteres. Pero, con ella en medio, cuando se producen menos de 4 coincidencias, el algoritmo destina esos 2 bytes para representarla, perdiendo bits y en consecuencia sumando tamaño al archivo final en detrimento de su reducción. Solamente a partir de la 5° coincidencia, el algoritmo estaría efectivamente comprimiendo. Por los resultados obtenidos, el balance fue a favor de las coincidencias menores a 4 caracteres y por ese motivo la tasa de compresión dio negativa.

Aparece la paradoja de la sábana corta. Si fuera posible aumentar el tamaño del *buffer de búsqueda* a 512 bytes, para incrementar la probabilidad de coincidencias de mayor longitud, seguramente se obtendría una mejor tasa de compresión. El problema radica en que, para representar 1024 posiciones de *offset* es necesario usar 10 bits y la única forma sería robando bits destinados a representar la *longitud coincidente*. Un bit menos para tal fin, restringiría al algoritmo a sólo detectar patrones de no más de 4 caracteres e impidiendo compresión alguna. La alternativa de emplear un tercer byte a la salida no es viable dado a que el algoritmo recién empezaría a comprimir a partir de la 7° coincidencia, algo muy poco probable de encontrar.

6.4.CONCLUSIONES

La ausencia de lógica en los datos de entrada es el motivo por el cual se descarta el empleo de los algoritmos de compresión sin pérdida basados en *diccionarios* para el seguimiento genético del ganado. Esto se observó en la implementación del algoritmo LZ77 donde se obtuvo una tasa de compresión desfavorable por la ausencia de coincidencias largas. Encontrar patrones más largos se logra incorporando más datos de entrada al

diccionario, mediante el aumento de su tamaño. Esto implica usar más memoria RAM de la que dispone la red WSN y la necesidad de que las coincidencias sean aún más grandes de lo previsto antes del nuevo *diccionario*, para compensar la adición de bytes en la terna de salida, usados para representar las nuevas posiciones del mismo.

Lo anterior no significa que la ciencia de compresión debería dejarse de considerar para una red WSN cuya aplicación sea diferente. Con esto se hace referencia que la inviabilidad de aplicar un algoritmo de compresión se debe pura y exclusivamente a dos factores principales: la cantidad de recursos disponibles y la conformación de los datos de entrada. El primero hace referencia principalmente a la memoria RAM y el segundo a que la interrelación de los datos de entrada tiene mucho o poco que ver con la tasa de compresión de acuerdo al algoritmo. Uno basado en *diccionario*, necesita que haya una estructura lógica en los datos a comprimir para encontrar coincidencias más largas. En cambio, si no fuera por el excesivo uso de RAM, el algoritmo de Huffman sería una alternativa interesante. Si se elige correctamente la unidad mínima de los datos de entrada, la relación inversa del algoritmo para codificar respecto a la frecuencia, puede tener mejor influencia en la tasa de compresión en aquellos datos donde no se presenta una interrelación fuerte.

El estudio llevado a acabo se orientó a la búsqueda de algoritmos de compresión que mejor se adapten al hardware disponible. Otro enfoque que podría usarse, es la adaptación de los algoritmos de compresión, con un procesamiento de firmware adicional, que supla las limitaciones de hardware. Este enfoque sugiere, por ejemplo, no descartar el algoritmo LZ77 por el excesivo uso de RAM, sino, suplantar esa necesidad con lecturas periódicas, que levanten de la memoria flash bloques más chicos del *diccionario*. Este procesamiento extra deberá cuidarse de no contrarrestar la tasa de compresión ganada con el aumento del tamaño del código a ejecutarse y además, no deberá olvidarse de las limitaciones energéticas.

A pesar de descartar la implementación de algoritmos en la red WSN para el Seguimiento Genético de Ganado, debe mencionarse que el uso de la compresión está ampliamente difundida en todos los medios, por lo que sería prudente aclarar que muchos algoritmos no ingresaron en el análisis de este trabajo final. Así, surge la necesidad de recomendar la continuación de la investigación y además sugerir un camino a seguir.

Una simple modificación en el algoritmo LZ77, muy usada en la mayoría de las variaciones del algoritmo, es eliminar la situación donde se usa una terna para codificar un simple carácter no comprimido. Usar una terna es poco eficiente, especialmente si muchos de los caracteres ocurren poco frecuentemente. La modificación consiste en agregar un bit de bandera, para indicar si lo que sigue es una terna codificada de un simple carácter no comprimido o una terna codificada debido a una compresión llevada a cabo. Así, la terna no

necesita más representar el carácter que sigue a la secuencia comprimida. Sólo será necesario mandar un par de valores correspondientes al *offset* y a la longitud de la coincidencia. Esta adaptación del algoritmo LZ77 fue englobada bajo el nombre de algoritmo LZSS por ser James Storer y Thomas Szymanski los autores en 1982 [11].

Esta adaptación es aún más favorable para nuestro caso particular, ya que se dispondría de 3 bits más para representar un *offset* mayor y así, emplear un *diccionario* más largo que permita aumentar la probabilidad de encontrar coincidencias más largas. Aun así, la limitada memoria RAM (512 bytes) de la plataforma WSN dificulta la implementación directa de este algoritmo debido a que un mayor *diccionario* implica manejar una *ventana deslizante* mayor.

Finalmente, recordemos que este trabajo orientó la gestión de la memoria únicamente en el del nodo Hijo. La memoria del nodo Madre, dado a que la comunicación es unidireccional, no almacena *muestras* y dispone de espacio adicional. De esta manera, por qué no dotar de mayor capacidad de procesamiento al nodo Hijo para que pueda retransmitir, las *muestras* que no pueda almacenar, al nodo transmisor. Nuevamente, no debe olvidarse que el funcionamiento del transceptor conlleva un consumo energético considerable y de una magnitud comparable con el acceso a escritura en la memoria flash.

7.IMPACTO AMBIENTAL

El impacto ambiental se refiere a los efectos sobre el medio ambiente que esta tecnología produce desde su fabricación hasta su último destino, al final de su ciclo de vida. Es de público conocimiento el gran incremento del consumo de aparatos electrónicos por las facilidades y beneficios que brindan, volviéndose necesario prever acciones para el tratamiento de la basura electrónica.

En la actualidad, las universidades e instituciones gubernamentales están llevando a cabo políticas de recolección y reciclaje de residuos electrónicos en centros vecinales de la Ciudad de Córdoba. De esta manera, la tecnología presentada en este trabajo, tiene un destino más alentador que el de los basurales a cielo abierto que hoy en día pueblan y arruinan el paisaje urbano.

8.RSU

La producción ganadera a pequeña, mediana y gran escala enfrenta el desafío de encontrar la línea genética de su ganado mediante métodos de bajo costo pero con asistencia humana, como la mera observación; o mediante métodos muy precisos pero de excesivo costo, como el análisis genético; o mediante métodos económicamente accesibles pero al muy largo plazo, como los sistemas RFID (Radio Frequency Identification). La tecnología WSN, propuesta en este trabajo, logra un balance favorable entre tiempo, costo y precisión frente a los métodos convencionales, ofreciendo al productor una nueva alternativa al seguimiento genético de ganado.

9.CONCLUSIONES

En este trabajo final, se implementó un firmware que gestiona, de una manera optimizada, el almacenamiento de *muestras* recibidas por la aplicación de seguimiento genético de ganado, en la escasa memoria flash disponible en el microcontrolador MSP430F2254 de la plataforma desarrollada por el LCRS. Este fue programado bajo el concepto de abstracción de hardware, con la finalidad de adaptarse a cualquier plataforma en caso de ser necesario.

El método de gestión elegido fue aquel que permitió una autonomía relativa por segmento mayor respecto a valores de autonomía de referencia ideales (56,88 *ciclos* y 170,66 *muestras* por segmento), que no contemplaban bits o bytes que cualquier algoritmo requiere para poder llevar a cabo la gestión básica de una memoria. Se obtuvo una autonomía entre el 89% y el 99% de 170,66 *muestras* por segmento de acuerdo a tres escenarios planteados. El impacto de sucesivos *ciclos vacíos* fue mitigado empleando encabezados de dos bits al inicio de cada *muestra* o cada byte de registro de *ciclos vacíos*, que identifican el *ciclo* y el tipo de información que le procede. Esto último se observó en los elevados porcentajes calculados de autonomía relativa de *ciclos*: 99,03 % al 356,54 % de 56,88 *ciclos* ideales.

Se creó una librería en lenguaje c basada en éste método, para la gestión integral de la memoria flash y cuyos servicios se orientaron al seguimiento genético de ganado.

Además, se investigó la conveniencia de la implementación de un algoritmo de compresión en la red WSN. Se estudiaron el algoritmo de Huffman y el algoritmo LZ77. La implementación del LZ77, condujo a concluir la inviabilidad de los mismos. Los principales motivos fueron la escases de memoria RAM para el almacenamiento temporal de *diccionarios* o tablas necesarias para la ejecución de los algoritmos y la ausencia de lógica en los datos de entrada que condujeron a tasas de compresión desfavorables del orden de las centenas de bytes. Se sugirió un camino a seguir para continuar la búsqueda de un algoritmo propicio para el seguimiento genético de ganado debido a la gran penetración de los mismos en todos los ámbitos.

10.REFERENCIAS

[1] C. M. Dwyer, "Genetic and physiological determinants of maternal behavior and lamb survival: Implications for low-imput sheep management.", Journal of animal science 86.14_supp, 2008.

[2] H. Karl y A. Willig, "Protocols and architectures for wireless sensor networks", John Wiley & Sons, 2005.

[3] M. Tubaishat, S. K. Madria, "Sensor networks: an overview", vol. 22, IEEE *Potentials*, 2003, pp. 20-23.

[4] J. Polastre, R. Szewczyk, "Analysis of wireless sensor networks for habitat monitoring", in: C. S. Raghavendra, K. M. Sivalingam, T. Znati (Eds), *Wireless Sensor Networks*, Springer, United States, 2004, pp. 399-423.

[5] K. Romer, F. Mattern, "The design space of wireless sensor networks", vol. 11, IEEE *Wireless Comunication*, 2004, pp. 54-61.

[6] P. Huang, L. Xiao, S. Soltani, M. W. Mutka, N. Xi, "The evolution of MAC protocols in wireless sensor networks: A survey, vol. 15, IEEE *Communications Surveys & Tutorials*, 2013, pp. 101-120.

[7] S. S. Alwakeel, N. A. Al-Nabhan, "A cooperative learning scheme for energy efficient routing in wireless sensor network", vol. 2, *Proc. of 11th International Conference on Machine Learning and Applications*, 2012, pp. 463-468.

[8] C. M. Vigorito, D. Ganesan, A. G. Barto, "Adaptive control of duty cycling in energy-harvesting wireless sensor networks", *Proc- of 4th Annual IEEE Communications Society Conference on Sensor; Mesh and Ad Hoc Communications and Networks*, 2007, pp. 21-30.

[9] E. Mandado, "Sistemas electrónicos digitales", 8 ed., vol. 1, Barcelona: Marcombo SA., 1998.

[10] K. Sayood, "Introduction to data compression", Nebraska: Elsevier Inc., 2006.

[11] Departamento de Ciencia y Tecnología Universidad Nacional de Quilmes, "Comunicación de datos", Buenos Aires.